

SUnet Reference Manual

For SUnet release 2.0
April 2003

Dr. S², Martin Gasbichler, Eric Marsden, Andreas Bernauer

Contents

Contents	2
1 Overview	5
1.1 Obtaining the system	6
1.2 How to use the packages	6
2 HTTP server	7
2.1 Starting and configuring the server	7
2.2 Requests	9
2.3 Responses	10
2.4 Response Bodies	12
2.5 Request Handlers	12
2.5.1 Basic Request Handlers	13
2.5.2 Static Content Request Handlers	14
2.6 CGI Server	17
2.7 Scheme-Evaluating Request Handlers	17
2.7.1 The loser structure	18
2.7.2 The toothless structure	18
2.7.3 The toothless-eval structure	18
2.8 Writing Request Handlers	19
2.8.1 Parsing HTML Forms	19
3 Parsing and Processing URIs	20
3.1 Notes on URI Syntax	20
3.2 Procedures	20

4	Parsing and Processing URLs	24
4.1	Server Records	24
4.2	HTTP URLs	25
5	Writing CGI Scripts in Scheme	27
6	FTP Server	28
7	FTP Client	31
8	Parsing Netrc Files	34
9	RFC 822 Library	36
10	Time and Daytime	38
10.1	Daytime	38
10.2	Time	38
11	SMTP Client	40
12	POP3 Client	43
13	DNS Client Library	45
13.1	Overview	45
13.2	Conditions	45
13.3	High-level Interface	47
13.4	Low-level Interface	48
13.5	Parsing /etc/resolv.conf	53
13.6	IP Addresses as Dotted Strings	53
	Index	55

Of course, there is no Underground—or Untergrund, as those German new-age kids like to call the movement whose orders they have sworn to follow. The age we all remember—the cliff-green turbocharged convertibles, cigarettes hanging loose in the corners of our mouths, and those trigger-happy fingers always ready for the quick hack—is long gone.

In retrospect, it all seems like a candy-colored dream, and it may very well be—after all, there was never any proof that the Untergrund ever existed, and even if it did, we can be sure the obedient followers of the shadowy movement leaders have long burned the papers, subjected the hard drives and diskettes to interminable sessions of the junkyard magnet, and eradicated all shreds of memory from the brains of those who might have talked through long sessions of Tcl hacking to the sounds of Celine Dion records.

Yet there are those who still covet membership in that secret cult—to gain access to its powerful lore, to usurp invidious and powerful superiors, or simply to impress their girlfriends. For those lost souls of the modern age, I have a few words of advice:

It's not a question of "membership"—silly merchandise and ridiculous certificates. If you are truly meant to be part of the Untergrund, you will know. *The Untergrund will find you.*

Alas, probably not.

April, 2003

Chapter 1

Overview

The Scheme Underground Networking Package (*SUnet*, for short) is a collection of applications and libraries for Internet hacking in Scheme. It runs under Scsh, the Scheme shell. SUnet includes the following components:

The SUnet Web server This is a highly configurable HTTP 1.0 server in Scheme. The server is accompanied by some libraries which may also be used separately:

- URI and URL parsers and unparsers
- a library for writing CGI scripts in Scheme
- server extensions for interfacing to CGI scripts
- server extensions for uploading Scheme code
- simple structured HTML output library

The SUnet ftp server This is a complete anonymous ftp server in Scheme.

ftp client library This library allows you to access ftp servers programmatically.

netrc library This library parses authentication information contained in `~/.netrc`.

SMTP client library This library allows you to forge mail from the comfort of your own Scheme process.

POP3 client library This library allows you to access your POP3 mailbox from inside scsh.

RFC822 header library This library parses email-style headers.

Daytime and Time protocol client libraries These libraries lets you find out what time it is without paying for a Rolex.

DNS client library This is a complete, multithreaded DNS library.

An `ls` clone This library displays Unix-style directory listings without running `ls`.

1.1 Obtaining the system

The SUnet code is available from <http://www.scsb.net/resources/sunet.html>. To run the code, you need version 0.6.4 or later of `scsh` from <http://www.scsb.net/>.

1.2 How to use the packages

Untar the SUnet distribution somewhere. Fire up `scsh` and load the SUnet `packages.scm` file into the configuration package. After that, all structures defined by SUnet are available:

```
atari-2600[72] scsh
Welcome to scsh 0.6.4 (...)
Type ,? for help.
> ,config ,load packages.scm
modules.scm
> ,open ftp
Load structure ftp (y/n)? y
[netrc netrc.scm]
[ftp ftp.scm]
> call library code
> ,exit
atari-2600[73]
```

Chapter 2

HTTP server

The SUnet HTTP Server is a complete industrial-strength implementation of the HTTP 1.0 protocol. It is highly configurable and allows the writing of dynamic web pages that run inside the server without going through complicated and slow protocols like CGI or Fast/CGI.

2.1 Starting and configuring the server

All procedures described in this section are exported by the `httpd` structure.

The Web server is started by calling the `httpd` procedure, which takes one argument, an options value:

`(httpd options)` \longrightarrow *no return value* procedure

This procedure starts the server. The *options* argument specifies various configuration parameters, explained below.

The server's basic loop is to wait on the port for a connection from an HTTP client. When it receives a connection, it reads in and parses the request into a special request data structure. Then the server forks a thread which binds the current I/O ports to the connection socket, and then hands off to the top-level request handler (which must be specified in the options). The request handler is responsible for actually serving the request—it can be any arbitrary computation. Its output goes directly back to the HTTP client that sent the request.

Before calling the request handler to service the request, the HTTP server installs an error handler that fields any uncaught error, sends an error reply to the client, and aborts the request transaction. Hence any error caused by a request handler will be handled in a reasonable and robust fashion.

The *options* argument can be constructed through a number of procedures with names of the form *with-...*. Each of these procedures either creates a fresh options value or adds a configuration parameter to an old options argument. The configuration parameter value is always the first argument, the (old) options value the optional second one. Here they are:

(with-port *port* [*options*]) → *options* procedure

This specifies the port on which the server listens. Defaults to 80.

(with-root-directory *root-directory* [*options*]) → *options* procedure

This specifies the current directory of the server. Note that this is *not* the document root directory. Defaults to */*.

(with-fqdn *fqdn* [*options*]) → *options* procedure

This specifies the fully-qualified domain name the server uses in automatically generated replies, or *#f* if the server should query DNS for the fully-qualified domain name.. Defaults to *#f*.

(with-reported-port *reported-port* [*options*]) → *options* procedure

This specifies the port number the server uses in automatically generated replies or *#f* if the reported port is the same as the port the server is listening on. (This is useful if you're running the server through an accelerating proxy.) Defaults to *#f*.

(with-server-admin *mail-address* [*options*]) → *options* procedure

This specifies the email address of the server administrator the server uses in automatically generated replies. Defaults to *#f*.

(with-request-handler *request-handler* [*options*]) → *options* procedure

This specifies the request handler of the server to which the server delegates the actual work. More on that subject below in Section 2.5. This parameter must be specified.

(with-simultaneous-requests *requests* [*options*]) → *options* procedure

This specifies a limit on the number of simultaneous requests the server servers. If that limit is exceeded during operation, the server will hold off on new requests until the number of simultaneous requests has sunk below the limit again. If this parameter is *#f*, no limit is imposed. Defaults to *#f*.

(with-log-file *log-file* [*options*]) → *options* procedure

This specifies the name of a log file for the server where it writes Common Log Format logging information. It can also be a port in which case the information is logged to that port, or `#f` for no logging. Defaults to `#f`.

To allow rotation of log files, the server re-opens the log file whenever it receives a `USR1` signal.

`(with-syslog? syslog? [options])` \longrightarrow *options* procedure

This specifies whether the server will log information about incoming to the Unix syslog facility. Defaults to `#t`.

`(with-resolve-ip? resolve-ip? [options])` \longrightarrow *options* procedure

This specifies whether the server writes the domain names rather than numerical IPs to the output log it produces. Defaults to `#t`.

To avoid paranthesis, the `make-httpd-options` procedure eases the construction of the options argument:

`(make-httpd-options transformer value ...)` \longrightarrow *options* procedure

This constructs an options value from an argument list of parameter transformers and parameter values. The arguments come in pairs, each an option transformer from the list above, and a value for that parameter. `Make-httpd-options` returns the resulting options value.

For example,

```
(httpd (make-httpd-options
  with-request-handler (rooted-file-handler "/usr/local/etc/httpd")
  with-root-directory "/usr/local/etc/httpd"))
```

starts the server on port 80 with `/usr/local/etc/httpd` as its root directory and lets it serve any file out from this directory.

2.2 Requests

Request handlers operate on *requests* which contain the information needed to generate a page. The relevant procedures to dissect requests are defined in the `httpd-requests` structure:

<code>(request? <i>value</i>)</code>	\longrightarrow <i>boolean</i>	procedure
<code>(request-method <i>request</i>)</code>	\longrightarrow <i>string</i>	procedure
<code>(request-uri <i>request</i>)</code>	\longrightarrow <i>string</i>	procedure
<code>(request-url <i>request</i>)</code>	\longrightarrow <i>url</i>	procedure
<code>(request-version <i>request</i>)</code>	\longrightarrow <i>pair</i>	procedure

`(request-headers request)` \longrightarrow *list* procedure
`(request-socket request)` \longrightarrow *socket* procedure

The procedure inspect request values. `Request?` is a predicate for requests. `Request-method` extracts the method of the HTTP request; it's a string such as "GET", "PUT". `Request-uri` returns the escaped URI string as read from request line. `Request-url` returns an HTTP URL value (see the description of the `url` structure in 4). `Request-version` returns (major . minor) integer pair representing the version specified in the HTTP request. `Request-headers` returns an association lists of header field names and their values, each represented by a list of strings, one for each line. `Request-socket` returns the the socket connected to the client.¹

2.3 Responses

A path handler must return a *response* value representing the content to be sent to the client. The machinery presented here for constructing responses lives in the `httpd-responses` structure.

`(make-response status-code maybe-message seconds mime extras body)` \longrightarrow *response* procedure

This procedure constructs a response value. *Status-code* is an HTTP status code (more on that below). *Maybe-message* is a a message elaborating on the circumstances of the status code; it can also be `#f` meaning that the server should send a default message associated with the status code. *Seconds* natural number indicating the time the content was created, typically the value of `(time)`. *Mime* is a string indicating the MIME type of the response (such as "text/html" or "application/octet-stream"). *Extras* is an association list with extra headers to be added to the response; its elements are pairs, each of which consists of a symbol representing the field name and a string representing the field value. *Body* represents the body of the response; more on that below.

`(make-redirect-response location)` \longrightarrow *response* procedure

This is a helper procedure for constructing HTTP redirections. The server will serve the new file indicated by *location*. *Location* must be URI-encoded and begin with a slash.

`(make-error-response status-code request [message] extras ...)` \longrightarrow *response* procedure

¹Request handlers should not perform I/O on the request record's socket. Request handlers are frequently called recursively, and doing I/O directly to the socket might bypass a filtering or other processing step interposed on the current I/O ports by some superior request handler.

This is a helper procedure for constructing error responses. *code* is status code of the response (see below). *Request* is the request that led to the error. *Message* is an optional string containing an error message written in HTML, and *extras* are further optional arguments containing further message lines to be added to the web page that's generated.

`Make-error-response` constructs a response value which generates a web page containing a short explanatory message for the error at hand.

ok	200	OK
created	201	Created
accepted	202	Accepted
prov-info	203	Provisional Information
no-content	204	No Content
mult-choice	300	Multiple Choices
moved-perm	301	Moved Permanently
moved-temp	302	Moved Temporarily
method	303	Method (obsolete)
not-mod	304	Not Modified
bad-request	400	Bad Request
unauthorized	401	Unauthorized
payment-req	402	Payment Required
forbidden	403	Forbidden
not-found	404	Not Found
method-not-allowed	405	Method Not Allowed
none-acceptable	406	None Acceptable
proxy-auth-required	407	Proxy Authentication Required
timeout	408	Request Timeout
conflict	409	Conflict
gone	410	Gone
internal-error	500	Internal Server Error
not-implemented	501	Not Implemented
bad-gateway	502	Bad Gateway
service-unavailable	503	Service Unavailable
gateway-timeout	504	Gateway Timeout

Table 2.1: HTTP status codes

<code>(status-code <name>)</code>	\longrightarrow	<i>status-code</i>	syntax
<code>(name->status-code symbol)</code>	\longrightarrow	<i>status-code</i>	procedure
<code>(status-code-number status-code)</code>	\longrightarrow	<i>integer</i>	procedure
<code>(status-code-message status-code)</code>	\longrightarrow	<i>string</i>	procedure

The `status-code` syntax returns a status code where *<name>* is the name

from Table 2.1. `Name->status-code` also returns a status code for a name represented as a symbol. For a given status code, `status-code-number` extracts its number, and `status-code-message` extracts its associated default message.

2.4 Response Bodies

A *response body* represents the body of an HTTP response. There are several types of response bodies, depending on the requirements on content generation.

`(make-writer-body proc)` \longrightarrow *body* procedure

This constructs a response body from a *writer*—a procedure that prints the page contents to a port. The *proc* argument must be a procedure accepting an output port (to which *proc* prints the body) and the options value passed to the `httpd` invocation.

`(make-reader-writer-body proc)` \longrightarrow *body* procedure

This constructs a response body from a *reader/writer*—a procedure that prints the page contents to a port, possibly after reading input from the socket of the HTTP connection. The *proc* argument must be a procedure accepting three arguments: an input port (associated with the HTTP connection socket), an output port (to which *proc* prints the body), and the options value passed to the `httpd` invocation.

2.5 Request Handlers

A request handler generates the actual content for a request; request handlers form a simple algebra and may be combined and composed in various ways.

A request handler is a procedure of two arguments like this:

`(request-handler path req)` \longrightarrow *response* procedure

Req is a request. The *path* argument is the URL's path, parsed and split at slashes into a string list. For example, if the Web client dereferences URL

`http://clark.lcs.mit.edu:8001/h/shivers/code/web.tar.gz`

then the server would pass the following path to the top-level handler:

`("h" "shivers" "code" "web.tar.gz")`

The *path* argument's pre-parsed representation as a string list makes it easy for the request handler to implement recursive operations dispatch on URL paths.

The request handler must return an HTTP response.

2.5.1 Basic Request Handlers

The web server comes with a useful toolbox of basic request handlers that can be used and built upon. The following procedures are exported by the `httpd-basic-handlers` structure:

`null-request-handler` `request-handler`

This request handler always generated a not-found error response, no matter what the request is.

`(make-predicate-handler predicate handler default-handler)` \longrightarrow `request-handler` procedure

The request handler returned by this procedure first calls *predicate* on its path and request; it then acts like *handler* if the predicate returned a true value, and like *default-handler* if the predicate returned `#f`.

`(make-host-name-handler hostname handler default-handler)` \longrightarrow `request-handler` procedure

The request handler returned by this procedure compares the host name specified in the request with *hostname*: if they match, it acts like *handler*, otherwise, it acts like *default-handler*.

`(make-path-predicate-handler predicate handler default-handler)` \longrightarrow `request-handler` procedure

The request handler returned by this procedure first calls *predicate* on its path; it then acts like *handler* if the predicate returned a true value, and like *default-handler* if the predicate returned `#f`.

`(make-path-prefix-handler path-prefix handler default-handler)` \longrightarrow `request-handler` procedure

This constructs a request handler that calls *handler* on its argument if *path-prefix* (a string) is the first element of the requested path; it calls *handler* on the rest of the path and the original request. Otherwise, the handler acts like *default-handler*.

`(alist-path-dispatcher handler-alist default-handler)` \longrightarrow `request-handler` procedure

This procedure takes as arguments an alist mapping strings to path handlers, and a default request handler, and returns a handler that dispatches on its path argument. When the new request handler is applied to a path

`("foo" "bar" "baz")`

it uses the first element of the path—`foo`—to index into the alist. If it finds an associated request handler in the alist, it hands the request off to that handler, passing it the tail of the path, in this case

`("bar" "baz")`

On the other hand, if the path is empty, or the alist search does not yield a hit, we hand off to the default path handler, passing it the entire original path,

```
("foo" "bar" "baz")
```

This procedure is how you say: “If the first element of the URL’s path is ‘foo’, do X; if it’s ‘bar’, do Y; otherwise, do Z.” The slash-delimited URI path structure implies an associated tree of names. The request-handler system and the alist dispatcher allow you to procedurally define the server’s response to any arbitrary subtree of the path space.

Example: A typical top-level request handler is

```
(define ph
  (alist-path-dispatcher
    ‘(("h"      . , (home-dir-handler "public.html"))
      ("cgi-bin" . , (cgi-handler "/usr/local/etc/httpd/cgi-bin"))
      ("seval"   . , seval-handler))
    (rooted-file-handler "/usr/local/etc/httpd/htdocs")))
```

This means:

- If the path looks like ("h" "shivers" "code" "web.tar.gz"), pass the path ("shivers" "code" "web.tar.gz") to a home-directory request handler.
- If the path looks like ("cgi-bin" "calendar"), pass ("calendar") off to the CGI request handler.
- If the path looks like ("seval" ...), the tail of the path is passed off to the code-uploading seval path handler.
- Otherwise, the whole path is passed to a rooted file handler, who will convert it into a filename, rooted at /usr/local/etc/httpd/htdocs, and serve that file.

2.5.2 Static Content Request Handlers

The request handlers described in this section are for serving static content off directory trees in the file system. They live in the `httpd-file-directory-handlers` structure.

The request handlers in this section eventually call an internal procedure named `file-serve` for serving files which implements a simple directory-generation service using the following rules:

- If the filename has the form of a directory (i.e., it ends with a slash), then `file-serve` actually looks for a file named `index.html` in that directory.

- If the filename names a directory, but is not in directory form (i.e., it doesn't end in a slash, as in `/usr/include` or `/usr/raj`), then `file-serve` sends back a "301 moved permanently" message, redirecting the client to a slash-terminated version of the original URL. For example, the URL `http://clark.lcs.mit.edu/ shivers` would be redirected to `http://clark.lcs.mit.edu/ shivers/`
- If the filename names a regular file, it is served to the client.

The `httpd-file-directory-handlers` all take an `options` value as an argument, similar to the options for `httpd` itself.

The `options` argument can be constructed through a number of procedures with names of the form `with-...`. Each of these procedures either creates a fresh `options` value or adds a configuration parameter to an old `options` argument. The configuration parameter value is always the first argument, the (old) `options` value the optional second one. Here they are:

`(with-file-name->content-type proc [options]) → options procedure`

This specifies a procedure for determining the MIME content type (`"text/html"`, `"application/octet-stream"` etc.) from a file name. *Proc* takes a file name as an argument and must return a string. (This is relevant in directory listings.) The default is a procedure able to handle the more common file extensions.

`(with-file-name->content-encoding proc [options]) → options procedure`

This specifies a procedure for determining the MIME content encoding (if the file is compressed, gzipped, etc.) from a file name. (This is relevant in directory listings.) *Proc* takes a file name as an argument and must return two values: the equivalent, unencoded file name (i.e., without the trailing `.Z` or `.gz`) and a string representing the content encoding.

`(with-file-name->icon-url proc [options]) → options procedure`

This specifies a procedure for determining the icon to be displayed next to a file name in a directory listing. *Proc* takes a file name as an argument and must return a URL for the corresponding icon or `#f`.

`(with-blank-icon-url file-name-or-#f [options]) → options procedure`

This specifies a file name (or its absence) for the special icon that must be as wide as the icons returned by the previous procedure but that is blank.

`(with-back-icon-url file-name-or-#f [options]) → options procedure`

This specifies a file name (or its absence) for the special icon that is displayed next to the "parent directory" link in directory listings.

(with-unknown-icon-url *file-name-or-#f* [*options*]) \longrightarrow *options* procedure

This specifies a file name (or its absence) for the special icon that is displayed next to the unknown entries in directory listings.

The `make-file-directory-options` procedure eases the construction of the options argument:

(make-file-directory-options *transformer value* ...) \longrightarrow *options* procedure

This constructs an options value from an argument list of parameter transformers and parameter values. The arguments come in pairs, each an option transformer from the list above, and a value for that parameter. `Make-file-directory-options` returns the resulting options value.

Here are procedure for constructing static content request handlers:

(rooted-file-handler *root* [*options*]) \longrightarrow *request-handler* procedure

This returns a request handler that serves files from a particular root in the file system. Only the GET operation is provided. The path argument passed to the handler is converted into a filename, and appended to *root*. The file name is checked for `..` components, and the transaction is aborted if it does. Otherwise, the file is served to the client.

(rooted-file-or-directory-handler *root* [*options*]) \longrightarrow *request-handler* procedure

Dito, but also serve directory indices for directories without `index.html`.

(home-dir-handler *subdir* [*options*]) \longrightarrow *request-handler* procedure

This procedure builds a request handler that does basic file serving out of home directories. If the resulting *request-handler* is passed a path of the form (*user . file-path*), then it serves the file *subdir/file-path* inside the user's home directory.

The request handler only handles GET requests; the filename is not allowed to contain `..` elements.

(tilde-home-dir-handler *subdir default-request-handler* [*options*]) \longrightarrow *request-handler* procedure

This returns request handler that examines the car of the path. If it is a string beginning with a tilde, e.g., " ziggy", then the string is taken to mean a home directory, and the request is served similarly to a `home-dir-handler` request handler. Otherwise, the request is passed off in its entirety to the *default-request-handler*.

2.6 CGI Server

The procedure(s) described here live in the `httpd-cgi-handlers` structure.

`(cgi-handler bin-dir [cgi-bin-path])` \longrightarrow *request-handler* procedure

Returns a request handler for CGI scripts located in *bin-dir*. *Cgi-bin-dir* specifies the value of the `PATH` variable of the environment the CGI scripts run in. It defaults to

`/bin:/usr/bin:/usr/ucb:/usr/bsd:/usr/local/bin`

The CGI scripts are called as specified by CGI/1.1².

Note that the CGI handler looks at the name of the CGI script to determine how it should be handled:

- If the name of the script starts with ‘`nph-`’, its reply is read, the RFC 822-fields like `Content-Type` and `Status` are parsed and the client is sent back a real HTTP reply, containing the rest of the script’s output.
- If the name of the script doesn’t start with ‘`nph-`’, its output is sent back to the client directly. If its return code is not zero, an error message is generated.

2.7 Scheme-Evaluating Request Handlers

The `httpd-seval-handlers` structure contains a handler which demonstrates how to safely evaluate Scheme code uploaded from the client to the server.

seval-handler request-handler

This request handler is suitable for receiving code entered into an HTML text form. The Scheme code being uploaded is being POSTed to the server (from a form). The code should be URI-encoded in the URL as `program=<stuff>`. *stuff* must be an (URI-encoded) Scheme expression which the handler evaluates in a separate subprocess. (It waits for 10 seconds for a result, then kills the subprocess.) The handler then prints the return values of the Scheme code.

The following structures define environments that are R5RS without features that could examine or effect the file system. You can also use them as models of how to execute code in other protected environments in Scheme 48.

²see <http://hoo.hoo.ncsa.uiuc.edu/cgi/interface.html> for a sort of specification.

2.7.1 The loser structure

The loser package exports only one procedure:

(loser *name*) \longrightarrow *nothing* procedure
Raises an error like “Illegal call *name*”.

2.7.2 The toothless structure

The toothless structure contains everything of R5RS except that following procedure cause an error if called:

- call-with-input-file
- call-with-output-file
- load
- open-input-file
- open-output-file
- transcript-on
- with-input-from-file
- with-input-to-file
- eval
- interaction-environment
- scheme-report-environment

2.7.3 The toothless-eval structure

(eval-safely *expression*) \longrightarrow *any result* procedure
Creates a brand-new structure, imports the toothless structure, and evaluates *expression* in it. When the evaluation is done, the environment is thrown away, so *expression*'s side-effects don't persist from one eval-safely call to the next. If *expression* raises an error exception, eval-safely returns #f.

2.8 Writing Request Handlers

2.8.1 Parsing HTML Forms

In HTML forms, field data are turned into a single string, of the form $\langle name \rangle = \langle val \rangle \& \langle name \rangle = \langle val \rangle \dots$. The `parse-html-forms` structure provides simple functionality to parse these strings.

`(parse-html-form-query string)` \longrightarrow *alist* procedure

This parses "foo=x&bar=y" into `((("foo" . "x") ("bar" . "y")))`. Substrings are plus-decoded (i.e. plus characters are turned into spaces) and then URI-decoded.

This implementation is slightly sleazy as it will successfully parse a string like "a&b=c&d=f" into `((("a&b" . "c") ("d" . "f")))` without a complaint.

Chapter 3

Parsing and Processing URIs

The `uri` structure contains a library for dealing with URIs.

3.1 Notes on URI Syntax

A URI (Uniform Resource Identifier) is of following syntax:

`[scheme] : path [? search] [# fragid]`

Parts in brackets may be omitted.

The URI contains characters like `:` to indicate its different parts. Some special characters are *escaped* if they are a regular part of a name and not indicators for the structure of a URI. Escape sequences are of following scheme: `%hh` where *h* is a hexadecimal digit. The hexadecimal number refers to the ASCII of the escaped character, e.g. `%20` is space (ASCII 32) and `%61` is 'a' (ASCII 97). This module provides procedures to escape and unescape strings that are meant to be used in a URI.

3.2 Procedures

`(parse-uri uri-string)` \longrightarrow `scheme path search frag-id` procedure

Parses an *uri-string* into its four fields. The fields are *not* unescaped, as the rules for parsing the *path* component in particular need unescaped text, and are dependent on *scheme*. The URL parser is responsible for doing this. If the *scheme*, *search* or *fragid* portions are not specified, they are `#f`. Otherwise, *scheme*, *search*, and *fragid* are strings. *path* is a non-empty string list—the path split at slashes.

Here is a description of the parsing technique. It is inwards from both ends:

- First, the code searches forwards for the first reserved character (=, ;, /, #, ?, : or space). If it's a colon, then that's the *scheme* part, otherwise there is no *scheme* part. At all events, it is removed.
- Then the code searches backwards from the end for the last reserved char. If it's a sharp, then that's the *fragid* part—remove it.
- Then the code searches backwards from the end for the last reserved char. If it's a question-mark, then that's the *search* part—remove it.
- What's left is the path. The code split it at slashes. The empty string becomes a list containing the empty string.

This scheme is tolerant of the various ways people build broken URI's out there on the Net¹, e.g. = is a reserved character, but used unescaped in the search-part. It was given to me² by Dan Connolly of the W3C and slightly modified.

(unescape-uri *string* [*start*] [*end*]) → *string* procedure

Unescape-uri unescapes a string. If *start* and/or *end* are specified, they specify start and end positions within *string* should be unescaped.

This procedure should only be used *after* the URI was parsed, since unescaping may introduce characters that blow up the parse—that's why escape sequences are used in URIs.

uri-escaped-chars char-set

This is a set of characters (in the sense of SRFI 14) which are escaped in URIs. These are the following characters: \$, -, _, @, ., &, !, *, \, ", ', (,), ,, +, and all other characters that are neither letters nor digits (such as space and control characters).

(escape-uri *string* [*escaped-chars*]) → *string* procedure

This procedure escapes characters of *string* that are in *escaped-chars*. *Escaped-chars* defaults to uri-escaped-chars.

Be careful with using this procedure to chunks of text with syntactically meaningful reserved characters (e.g., paths with URI slashes or colons)—they'll be escaped, and lose their special meaning. E.g. it would be a mistake to apply escape-uri to

//lcs.mit.edu:8001/foo/bar.html

¹So it does not absolutely conform to RFC 1630.

²That's Olin Shivers.

because the slashes and colons would be escaped.

`(split-uri uri start end)` \longrightarrow *list* procedure

This procedure splits *uri* at slashes. Only the substring given with *start* (inclusive) and *end* (exclusive) as indices is considered. *start* and *end* - 1 have to be within the range of *uri*. Otherwise an `index-out-of-range` exception will be raised.

Example:

```
(split-uri "foo/bar/colon" 4 11)
returns
("bar" "col")
```

`(uri-path->uri path)` \longrightarrow *string* procedure

This procedure generates a path out of a URI path list by inserting slashes between the elements of *plist*.

If you want to use the resulting string for further operation, you should escape the elements of *plist* in case they contain slashes, like so:

```
(uri-path->uri (map escape-uri pathlist))
```

`(simplify-uri-path path)` \longrightarrow *list* procedure

This procedure simplifies a URI path. It removes "." and "../" entries from path, and removes parts before a root. The result is a list, or `#f` if the path tries to back up past root.

According to RFC 2396, relative paths are considered not to start with /. They are appended to a base URL path and then simplified. So before you start to simplify a URL try to find out if it is a relative path (i.e. it does not start with a /).

Examples:

```
(simplify-uri-path (split-uri "/foo/bar/baz/.." 0 15))
⇒ (" " "foo" "bar")

(simplify-uri-path (split-uri "foo/bar/baz/../../../../" 0 20))
⇒ ()

(simplify-uri-path (split-uri "/foo/../../../../" 0 10))
⇒ #f

(simplify-uri-path (split-uri "foo/bar//" 0 9))
```

\Rightarrow `()`

`(simplify-uri-path (split-uri "foo/bar/" 0 8))`
 \Rightarrow `()`

`(simplify-uri-path (split-uri "/foo/bar//baz/../../" 0 19))`
 \Rightarrow `#f`

Chapter 4

Parsing and Processing URLs

This module contains procedures to parse and unparse URLs. Until now, only the parsing of HTTP URLs is implemented.

4.1 Server Records

A *server* value describes path prefixes of the form *user:password@host:port*. These are frequently used as the initial prefix of URLs describing Internet resources.

(make-server <i>user password host port</i>)	→ <i>server</i>	procedure
(server? <i>thing</i>)	→ <i>boolean</i>	procedure
(server-user <i>server</i>)	→ <i>string-or-#f</i>	procedure
(server-password <i>server</i>)	→ <i>string-or-#f</i>	procedure
(server-host <i>server</i>)	→ <i>string-or-#f</i>	procedure
(server-port <i>server</i>)	→ <i>string-or-#f</i>	procedure

Make-server creates a new server record. Each slot is a decoded string or #f. (*Port* is also a string.)

server? is the corresponding predicate, server-user, server-password, server-host and server-port are the corresponding selectors.

(parse-server <i>path default</i>)	→ <i>server</i>	procedure
(server->string <i>server</i>)	→ <i>string</i>	procedure

Parse-server parses a URI path *path* (a list representing a path, not a string) into a server value. Default values are taken from the server *default* except for the host. The values are unescaped and stored into a server record that is returned. Fatal-syntax-error is called, if the specified path has no initial slashes (i.e., it starts with '/...').

`server->string` just does the inverse job: it unparses *server* into a string. The elements of the record are escaped before they are put together.

Example:

```
> (define default (make-server "andreas" "se ret" "www.sf.net" "80"))
> (server->string default)
"andreas:se%20ret@www.sf.net:80"
> (parse-server '(" " " "foo%20bar@www.scsb.net" "docu" "index.html")
  default)
'#server
> (server->string ##)
"foo%20bar:se%20ret@www.scsb.net:80"
```

For details about escaping and unescaping see Chapter 3.

4.2 HTTP URLs

<code>(make-http-url <i>server path search frag-id</i>)</code>	\longrightarrow <i>http-url</i>	procedure
<code>(http-url? <i>thing</i>)</code>	\longrightarrow <i>boolean</i>	procedure
<code>(http-url-server <i>http-url</i>)</code>	\longrightarrow <i>server</i>	procedure
<code>(http-url-path <i>http-url</i>)</code>	\longrightarrow <i>list</i>	procedure
<code>(http-url-search <i>http-url</i>)</code>	\longrightarrow <i>string-or-#f</i>	procedure
<code>(http-url-fragment-identifier <i>http-url</i>)</code>	\longrightarrow <i>string-or-#f</i>	procedure

`Make-http-url` creates a new `httpd-url` record. *Server* is a record, containing the initial part of the address (like `anonymous@clark.lcs.mit.edu:80`). *Path* contains the URL's URI path (a list). These elements are in raw, unescaped format. To convert them back to a string, use `(uri-path-list->path (map escape-uri pathlist))`. *Search* and *frag-id* are the last two parts of the URL. (See Chapter 3 about parts of an URI.)

`Http-url?` is the predicate for HTTP URL values, and `http-url-server`, `http-url-path`, `http-url-search` and `http-url-fragment-identifier` are the corresponding selectors.

<code>(parse-http-url <i>path search frag-id</i>)</code>	\longrightarrow <i>http-url</i>	procedure
<code>(http-url->string <i>http-url</i>)</code>	\longrightarrow <i>string</i>	procedure

This constructs an HTTP URL record from a URI path (a list of path components), a search, and a frag-id component.

`Http-url->string` just does the inverse job. It converts an HTTP URL record into a string.

Note: The URI parser `parse-uri` maps a string to four parts: *scheme*, *path*, *search* and *frag-id* (see Section 3.2 for details). If *scheme* is `http`, then the

other three parts can be passed to `parse-http-url`, which parses them into a `http-url` record. All strings come back from the URI parser encoded. *Search* and *frag-id* are left that way; this parser decodes the path elements. The first two list elements of the path indicating the leading double-slash are omitted.

The following procedure combines the jobs of `parse-uri` and `parse-http-url`:

`(parse-http-url-string string)` \longrightarrow *http-url* procedure

This parses an HTTP URL and returns the corresponding URL value; it calls `fatal-syntax-error` if the URL string doesn't have an `http` scheme.

Chapter 5

Writing CGI Scripts in Scheme

The `cgi-scripts` structure provides functionality useful for writing CGI scripts in Scheme.

`(cgi-form-query)` \longrightarrow *data-alist* procedure

CGI scripts receive their parameters in various ways, depending on how they were called (e.g. by GET method).

This procedure translates the delivered form data into an alist of decoded strings, using the environment variables set by the server (`REQUEST_METHOD`, `QUERY_STRING` (for a GET request), `CONTENT_LENGTH` (for a POST request)). So a query string like

`button=on&reply=0h,%20yes`

becomes an alist

`((("button" . "on") ("reply" . "0h, yes")))`

`Cgi-form-query` only works for GET and POST methods.

Chapter 6

FTP Server

The `ftpd` structure contains a complete anonymous ftp server.

(`ftpd options`) \longrightarrow *no return value* procedure

(`ftp-inetd options`) \longrightarrow *no return value* procedure

`Ftpd` starts the server, using *anonymous-home* as the root directory of the server.

`ftp-inetd` is the version to be used from `inetd`. `Ftpd-inetd` handles the connection through the current standard output and input ports.

The *options* argument can be constructed through a number of procedures with names of the form `with-...`. Each of these procedures either creates a fresh options value or adds a configuration parameter to an old options argument. The configuration parameter value is always the first argument, the (old) options value the optional second one. Here they are:

(`with-port port [options]`) \longrightarrow *options* procedure

This specifies the port on which the server listens. Defaults to 21.

(`with-anonymous-home string [options]`) \longrightarrow *options* procedure

This specifies the home directory for anonymous logins. Defaults to `"~ftp"`.

(`with-banner list [options]`) \longrightarrow *options* procedure

This specifies an alternative greeting banner for those members of the Untergrund who prefer to remain covert. The banner is represented as a list of strings, one for each line of output.

(`with-log-port output-port [options]`) \longrightarrow *options* procedure

If this is non-#f, exftpd outputs a log entry for each file sent or retrieved on *output-port*. Defaults to #f.

(with-dns-lookup? *boolean* [*options*]) \longrightarrow *options* procedure

If *dns-lookup?* is #t, the log file will contain the host names instead of their IP addresses. If *dns-lookup?* is #f, the log will only contain IP addresses. Defaults to #f.

The `make-ftp-options` eases the construction of the options argument:

(make-ftp-options *transformer value* ...) \longrightarrow *options* procedure

This constructs an options value from an argument list of parameter transformers and parameter values. The arguments come in pairs, each an option transformer from the list above, and a value for that parameter. `Make-ftp-options` returns the resulting options value.

The log format of `ftpd` is the same as the one of `wuftpd`. The entries look like this:

```
Fri Apr 19 17:08:14 2002 4 134.2.2.171 56881 /files.lst b _ i a nop@ssword ftp 0 *
```

These are the fields:

1. Current date and time. This field contains spaces and is 24 characters long.
2. Transfer time in seconds.
3. Remote host IP (`wu-ftp` puts the name here).
4. File size in bytes
5. Name of file (spaces are converted to underscores)
6. Transfer type: `ascii` or `binary` (image type).
7. Special action flags. As `ftpd` does not support any special action, it always has `_` here.
8. File was sent to user (`outgoing`) or received from user (`incoming`)
9. `A`nonymous access
10. Anonymous ftp password.
11. Service name—always `ftp`.
12. Authentication mode (always “none” = ‘0’).

13. Authenticated user ID (always “not available” = ‘*’)

The server also writes log information to the syslog facility. The following syslog levels occur in the output:

- notice
- messages concerning *connections* (establishing connection, connection refused, closing connection due to timeout, etc.)
 - the execution of the STOR command
Its success (*i.e.* somebody is putting something on your server via ftp, also known as PUT) is also logged at notice.
 - internal errors
 - Unix errors
 - reaching of actually unreachable case branches
- info Messages concerning all other commands, including the RETR command.
- debug all other messages, including debug messages

FTP Client

Some of the procedures in this module extract useful information from the server's reply, such as the size of a file, or the name of the directory we have moved to. These procedures return the extracted information, or, if the server's response doesn't match the expected code from the server, a catchable `ftp-error` is raised.

Open a command connection with the remote machine *host* and login on that server with *login* and *password*. *Login* and *password* can be `#f`, in which case the information is extracted from the user's `.netrc` file if necessary.

If *log-port* is specified, it must be an output port: this starts logging the conversation with the server to that port. Note that the log contains passwords in clear text.

This change the transfer mode for future file transfers. The transfer mode is specified by *ftp-type* which can be created with the `ftp-type` macro. *<Name>* must be either `binary` for binary data or `ascii` for text.

This changes the name of *old* on the remote host to *new* (assuming sufficient permissions). *Old* and *new* are strings.

(ftp-delete *connection file*) → *undefined* procedure
This deletes *file* from the remote host (assuming the user has appropriate permissions).

(ftp-cd *connection dir*) → *undefined* procedure
This changes the current directory on the server.

(ftp-cdup *connection*) → *undefined* procedure
This move to the parent directory on the server.

(ftp-pwd *connection*) → *string* procedure
Return the current directory on the remote host, as a string.

(ftp-ls *connection [dir]*) → *list* procedure
This returns a list of filenames on the remote host, either from the current directory (if *dir* is not specified), or from the directory specified by *dir*.

(ftp-dir *connection [dir]*) → *status* procedure
This returns a list of long-form file name entries on the remote host, either from the current directory (if *dir* is not specified), or from the directory specified by *dir*. (Note that the format for the long-form entries is not specified by the FTP standard.)

(ftp-get *connection remote-file proc*) → *undefined* procedure
This downloads *remote-file* from the FTP server. Ftp-get establishes a data conneciton to the server, attaches an input port to the data connection, and calls *proc* on that port.

(ftp-put *connection remote-file proc*) → *undefined* procedure
This uploads *remote-file* to the FTP server. Ftp-put establishes a data connection to the server, attaches an output port to the data connection, and calls *proc* on that port.

(ftp-append *connection remote-file proc*) → *undefined* procedure
This appends data to *remote-file* on the FTP server. Ftp-append establishes a data conneciton to the server, attaches an output port to the data connection, and calls *proc* on that port.

(ftp-rmdir *connection dir*) → *undefined* procedure
This removes the directory *dir* from the remote host (assuming sufficient permissions).

(ftp-mkdir *connection dir*) \longrightarrow *undefined* procedure

This create a new directory named *dir* on the remote host (assuming sufficient permissions).

(ftp-modification-time *connection file*) \longrightarrow *date* procedure

This requests the time of the last modification of *file* on the remote host, and on success return a Scsh date record. (This command is not part of RFC 959 and is not implemented by all servers, but is useful for mirroring.)

(ftp-size *connection file*) \longrightarrow *integer* procedure

This returns the size of *file* in bytes. (This command is not part of RFC 959 and is not implemented by all servers.)

(ftp-quit *connection*) \longrightarrow *undefined* procedure

This closes the connection to the remote host. The *connection* object is useless after a quit command.

(ftp-quot *connection command*) \longrightarrow *status* procedure

This sends a *command* verbatim to the remote server and wait for a response. The response text is returned verbatim.

(ftp-error? *thing*) \longrightarrow *boolean* procedure

This returns #t if *thing* is a ftp-error object, otherwise #f.

(copy-port->port-binary *input-port output-port*) \longrightarrow *undefined* procedure

(copy-port->port-ascii *input-port output-port*) \longrightarrow *undefined* procedure

(copy-ascii-port->port *input-port output-port*) \longrightarrow *undefined* procedure

These procedures are useful for downloading and uploading data to an FTP connection via ftp-get, ftp-get, and ftp-append. They all copy data from one port to another. Copy-port->port-binary copies verbatim, while the other two perform CR/LF conversion for ASCII data transfers. Copy-port->port-ascii adds CR/LFs at line endings on output, whereas Copy-ascii-port->port removes CR/LFs at line endings end replaces them by ordinary LFs.

Chapter 8

Parsing Netrc Files

The `netrc` structures provides procedures to parse authentication information contained in `/.netrc`.

On Unix systems the `netrc` file may contain information allowing automatic login to remote hosts. The format of the file is defined in the `ftp(1)` manual page. Example lines are

```
machine online.cict.fr login marsden password secret
default login anonymous password user@site
```

The `netrc` file should be protected by appropriate permissions, and (like `/usr/bin/ftp`) this library will refuse to read the file if it is badly protected. (unlike `ftp` this library will always refuse to read the file—`ftp` refuses it only if the password is given for a non-default account). Appropriate permissions are set if only the user has permissions on the file.

`(netrc-machine-entry host accept-default? [file-name])` \longrightarrow `netrc-entry-or-#f` procedure

This procedure looks for the entry related to given host in the user's `netrc` file. The host is specified in *host*. *Accept-default?* specifies whether `netrc-machine-entry` should fall back to the default entry if there is no match for *host* in the `netrc` file. If specified, *file-name* specifies an alternate file name for the `netrc` data. It defaults to `.netrc` in the current user's home directory.

`Netrc-machine-entry` returns a `netrc` entry (see below) if it was able to find the requested information; if not, it returns `#f`.

If the `netrc` file had inappropriate permissions, `netrc-machine-entry` raises an error.

<code>(netrc-entry? <i>thing</i>)</code>	\longrightarrow <i>boolean</i>	procedure
<code>(netrc-entry-machine <i>netrc-entry</i>)</code>	\longrightarrow <i>string</i>	procedure
<code>(netrc-entry-login <i>netrc-entry</i>)</code>	\longrightarrow <i>string-or-#f</i>	procedure
<code>(netrc-entry-password <i>netrc-entry</i>)</code>	\longrightarrow <i>string-or-#f</i>	procedure
<code>(netrc-entry-account <i>netrc-entry</i>)</code>	\longrightarrow <i>string-or-#f</i>	procedure

`Netrc-entry?` is the predicate for netrc entries. The other procedures are selectors for netrc entries as returned by `netrc-machine-entry`. They return `#f` if the netrc file didn't contain a binding for the corresponding field.

<code>(netrc-macro-definitions [<i>file-name</i>])</code>	\longrightarrow <i>alist</i>	procedure
---	--------------------------------	-----------

This returns the macro definitions from the netrc files, represented as an alist mapping macro names—represented as strings—to definitions—represented as lists of strings.

Chapter 9

RFC 822 Library

The `rfc822` structure provides rudimentary support for parsing headers according to RFC 822 *Standard for the format of ARPA Internet text messages*. These headers show up in SMTP messages, HTTP headers, etc.

An RFC 822 header field consists of a *field name* and a *field body*, like so:

```
Subject: RFC 822 can format itself in the ARPA
```

Here, the field name is ‘Subject’, and the field name is ‘ RFC 822 can format itself in the ARPA’ (note the leading space). The field body can be spread over several lines:

```
Subject: RFC 822 can format itself
       in the ARPA
```

In this case, RFC 822 specifies that the meaning of the field body is actually all the lines of the body concatenated, without the intervening line breaks.

The `rfc822` structure provides two sets of parsing procedures—one represents field bodies in the RFC-822-specified meaning, as a single string, the other (with `-with-line-breaks` appended to the names) reflects the line breaks and represents the bodies as a list of string, one for each line. The latter set only marginally useful—mainly for code that needs to output headers in the same form as they were originally provided.

`(read-rfc822-field [port] [read-line])` → *name body* procedure

`(read-rfc822-field-with-line-breaks [port] [read-line])` → *name body-lines* procedure

Read one field from the port, and return two values:

name This is a symbol describing the field name, such as `subject` or `to`. The symbol consists of all lower-case letters.¹

body or *body-lines* This is the field body. *Body* is a single string, *body-lines* is a list of strings, one for each line of the body. In each case, the terminating `cr/lf`'s (but nothing else) are trimmed from each string.

When there are no more fields—EOF or a blank line has terminated the header section—then both procedures returns `[#f #f]`.

Port is an optional input port to read from—it defaults to the value of `(current-input-port)`.

Read-line is an optional parameter specifying a procedure of one argument (the input port) used to read the raw header lines. The default used by these procedures terminates lines with either `cr/lf` or just `lf`, and it trims the terminator from the line. This procedure should trim the terminator of the line, so an empty line is returned as an empty string.

The procedure raises an error if the syntax of the read field (the line returned by the `read-line-function`) is illegal according to RFC 822.

`(read-rfc822-headers [port] [read-line])` → *alist* procedure
`(read-rfc822-headers-with-line-breaks [port] [read-line])` → *alist* procedure

This procedure reads in and parses a section of text that looks like the header portion of an RFC 822 message. It returns an association list mapping field names (a symbol such as `date` or `subject`) to field bodies. The representation of the field bodies is as with `read-rfc822-field` and `read-rfc822-field-with-line-breaks`.

These procedures preserve the order of the header fields. Note that several header fields might share the same field name—in that case, the returned *alist* will contain several entries with the same *car*.

Port and *read-line* are as with `read-rfc822-field` and `read-rfc822-field-with-line-breaks`.

`(rfc822-time->string time)` → *string* procedure

This formats a time value (as returned by `scsh`'s `time`) according to the requirements of the RFC 822 Date header field. The format looks like this:

Sun, 06 Nov 1994 08:49:37 GMT

¹In fact, it `read-rfc822-field` uses the preferred case for symbols of the underlying Scheme implementation which, in the case of `scsh`, happens to be lower-case.

Chapter 10

Time and Daytime

Many Unix hosts provide a RFC 867 Daytime service which sends the current date and time as a human-readable character string. The daytime service is typically served on port 13 as both TCP and UDP.

The RFC 868 Time protocol provides a site-independent, machine readable date and time. The Time service is typically served on port 37 as TCP and UDP. The idea is that you can confirm your system's idea of the time by polling several independent sites on the network.

10.1 Daytime

The `rfc867` structure contains an interface to Daytime protocol.

```
(rfc867-daytime/tcp host) → string           procedure  
(rfc867-daytime/udp host [timeout-or-#f]) → string-or-#f procedure
```

These procedures asks *host* about the current daytime and return the host's answer (e.g., "Thursday, April 4, 2").

`Rfc867-daytime/tcp` uses the TCP variant of the protocol. `Rfc867-daytime/udp` uses UDP and sends a single request to the server. It allows the specification of an optional timeout; if not specified or `#f`, `Rfc867-daytime/udp` will wait indefinitely for an answer. If the answer from the server doesn't arrive within the specified time, `rfc867-daytime/udp` returns `#f`.

10.2 Time

The `rfc868` structure contains an interface to the Time protocol.

```

(rfc868-time/tcp host)    → string           procedure
(rfc868-time/udp host [timeout-or-#f]) → string-or-#f procedure

```

These procedures asks *host* about the current time and return the host's answer. This is the number of second since 1970, just as with *scsh*'s *time* procedure.

rfc868-time/tcp uses the TCP variant of the protocol. *rfc868-time/udp* uses UDP and sends a single request to the server. It allows the specification of an optional timeout; if not specified or *#f*, *rfc868-time/udp* will wait indefinitely for an answer. If the answer from the server doesn't arrive within the specified time, *rfc868-time/udp* returns *#f*.

Chapter 11

SMTP Client

The `smtp` structure provides an client library for the Simple Mail Transfer Protocol, commonly used for sending email on the Internet. This library provides a simple wrapper for sending complete emails as well as procedures for composing custom SMTP transactions.

Some of the procedures described here return an SMTP reply code. For details, see RFC 821.

```
(smtp-send-mail from to-list headers body [host]) → undefined procedure
(smtp-error? thing) → boolean procedure
(smtp-recipients-rejected-error? thing) → boolean procedure
```

This emails message *body* with headers *headers* to recipients in list *to-list*, using a sender address *from*. The email is handed off to the SMTP server running on *host*; default is the local host. *Body* is either a list of strings representing the lines of the message body or an input port which is exhausted to determine the message body. *Headers* is an association lists, mapping symbols representing RFC 822 field names to strings representing field bodies.

If some transaction-related error happens, `smtp-send-mail` signals an `smtp-error` condition with predicate `smtp-error?`. More specifically, it raises an `smtp-recipients-rejected-error` (a subtype of `smtp-error`) if some recipients were rejected. For `smtp-error`, the arguments to the `signal` call are the error code and the error message, represented as a list of lines. For `smtp-recipients-rejected-error`, the arguments are reply code 700 and an association list whose elements are of the form (*loser-recipient code . text*)—that is, for each recipient refused by the server, you get the error data sent back for that guy. The success check is (`< code 400`).

<code>(smtp-expand <i>name host</i>)</code>	\longrightarrow	<code><i>code text</i></code>	procedure
<code>(smtp-verify <i>name host</i>)</code>	\longrightarrow	<code><i>code text</i></code>	procedure
<code>(smtp-get-help <i>host [details]</i>)</code>	\longrightarrow	<code><i>code text-list</i></code>	procedure

These three are simple queries of the server as stated in the RFC 821: `smtp-expand` asks the server to confirm that the argument identifies a mailing list, and if so, to return the membership of that list. The full name of the users (if known) and the fully specified mailboxes are returned in a multiline reply. `smtp-verify` asks the receiver to confirm that the argument identifies a user. If it is a user name, the full name of the user (if known) and the fully specified mailbox are returned. `smtp-get-help` causes the server to send helpful information. The command may take an argument (*details*) (e.g., any command name) and return more specific information as a response.

<code>(smtp-connect <i>host [port]</i>)</code>	\longrightarrow	<code><i>smtp-connection</i></code>	procedure
--	-------------------	-------------------------------------	-----------

`smtp-connect` returns an SMTP connection value that represents a connection to the SMTP server.

<code>(smtp-transactions <i>smtp-connection transaction1 ...</i>)</code>	\longrightarrow	<code><i>code text-list</i></code>	procedure
<code>(smtp-transactions/no-close <i>smtp-connection transaction1 ...</i>)</code>	\longrightarrow	<code><i>code text-list</i></code>	procedure

These procedures make it easy to do simple sequences of SMTP commands. *smtp-connection* must be an SMTP connection as returned by `smtp-connect`. The *transaction* arguments must be transactions as returned by the procedures below. `smtp-transactions` and `smtp-transactions/no-close` execute the transactions specified by the arguments.

For each transaction,

- If the transaction's reply code is 221 or 421 (meaning the socket has been closed), then the transaction sequence is aborted, and `smtp-transactions`/`smtp-transactions/no-close` return the reply code and text from that transaction.
- If the reply code is an error code (in the four- or five-hundred range), the transaction sequence is aborted, and the fatal transaction's code and text values are returned. `smtp-transactions` will additionally close the socket for you; `smtp-transactions/no-close` will not.
- If the transaction is the last in the transaction sequence, its reply code and text are returned.
- Otherwise, we throw away the current reply code and text, and proceed to the next transaction.

Smtplib closes the socket after the transaction. (The `smtp-quit` transaction, when executed, also closes the transaction.)

If the socket should be kept open in the case of an abort, use `Smtplib.no_close`.

<code>(smtp-helo local-host-name)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-mail sender-address)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-rcpt destination-address)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-data socket message)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-send sender-address)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-soml sender-address)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-saml sender-address)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>smtp-rset</code>			<code>smtp-transaction</code>
<code>(smtp-vrfy user)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-expn user)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>(smtp-help details)</code>	\longrightarrow	<code>smtp-transaction</code>	procedure
<code>smtp-noop</code>			<code>smtp-transaction</code>
<code>smtp-quit</code>			<code>smtp-transaction</code>
<code>smtp-turn</code>			<code>smtp-transaction</code>

These transactions represent the commands of the SMTP protocol for use in `smtp` and `smtp.no_close`, i.e. they send the corresponding command along with the argument(s), if any. For details, consult RFC 821.

The `smtp-quit` transaction, in addition to sending a QUIT command to the SMTP server, also closes the socket of its SMTP connection.

Chapter 12

POP3 Client

The `pop3` structure provides a client for the POP3 protocol that allows access to email on a maildrop server. It is often used in configurations where users connect from a client machine which doesn't have a permanent network connection or isn't always turned on, situations which make local SMTP delivery impossible. It is the most common form of email access provided by ISPs.

Two types of authentication are commonly used. The first, most basic type involves sending a user's password in clear over the network, and should be avoided. (Unfortunately, many POP3 clients only implement this basic authentication.) The digest authentication system involves the server sending the client a "challenge" token; the client encodes this token with the pass phrase and sends the coded information to the server. This method avoids sending sensitive information over the network. Both methods are implemented by `pop3`.

Once connected, a client may request information about the number and size of the messages waiting on the server, download selected messages (either their headers or the entire content), and delete selected messages.

The procedures defined here raise an error detectable via `pop3-error?` upon protocol errors with the POP3 server.

`(pop3-connect [host-or-#f] [login-or-#f] [password-or-#f] [log-port])` \longrightarrow *connection* procedure

This procedure connects to the maildrop server named *host*, and logs in using the provided login name and password. Any of these can be omitted or `#f`, in which case the procedure uses defaults: `MAILHOST` for the host, and `/ .netrc`-provided values for login and password. If *log-port* is provided, the conversation to the server is logged to the specified output port.

Pop3-connect returns a value representing the connection to the POP3 server, to be used in the procedures below.

(pop3-stat *connection*) \longrightarrow *number bytes* procedure

This returns the number of messages and the number of bytes waiting in the maildrop.

Most of the following procedures accept a *msgid* argument which specifies a message number, which ranges from 1 for the first message to the number returned by pop3-stat.

(pop3-retrieve-message *connection msgid*) \longrightarrow *headers message* procedure

This downloads message number *msgid* from the mailhost. It returns the headers as an alist of field names and bodies; the names are symbols, the bodies are strings. (These are obtained using the rfc822 structure, see Section 9.) The message is returned as a list of strings, each string representing a line of the message.

(pop3-retrieve-headers *connection msgid*) \longrightarrow *headers* procedure

This downloads the headers of message number *msgid*. It returns the headers in the same format as pop3-retrieve-message.

(pop3-last *connection*) \longrightarrow *msgid* procedure

This returns the highest accessed message-id number for the current session. (This isn't in the RFC, but seems to be supported by several servers.)

(pop3-delete *connection msgid*) \longrightarrow *undefined* procedure

This mark message number *msgid* for deletion. The message will not be deleted until the client logs out.

(pop3-reset *connection*) \longrightarrow *undefined* procedure

This marks any messages which have been marked for deletion.

(pop3-quit *connection*) \longrightarrow *undefined* procedure

This closes the connection with the mailhost.

(pop3-error? *thing*) \longrightarrow *boolean* procedure

This returns #t if *thing* is a pop3-error object, otherwise #f.

Chapter 13

DNS Client Library

Used files: `dns.scm`

Name of the package: `dns`

13.1 Overview

The `dns` structure contains a library for querying DNS servers. The library contains sophisticated replacements for `scsh`'s interface to the `gethostbyname` and `gethostbyaddr` and many extensions to these functions.

The main features of the library include:

- Complete implementation of the DNS protocol
- Concurrent contacting of multiple DNS servers without blocking the `scsh` process
- Internal caching of DNS responses
- Parsing of `resolv.conf`, including `search` entries to generate FQDNs from unqualified host names
- Rich condition hierarchy

13.2 Conditions

The library defines a set of conditions raised by the procedures of the library. The supertype of these conditions is `dns-error`.

`(dns-error? thing)` \longrightarrow *boolean* procedure

The predicate for dns-error conditions.

(dns-error->string *dns-error-condition*) \longrightarrow *string* procedure

Returns a string with the description of the condition.

parse-error	condition
unexpected-eof-from-server	condition
bad-address	condition
no-nameservers	condition
bad-nameserver	condition
not-a-hostname	condition
not-a-ip	condition

dns-format-error	condition
dns-server-failure	condition
dns-name-error	condition
dns-not-implemented	condition
dns-refused	condition

These conditons correspond to errors returned by the DNS server. They are all subtypes of the dns-server-error condition which in turn is a subtype of dns-error.

(dns-server-error? *thing*) \longrightarrow *boolean* procedure

The predicate for dns-server-error conditions.

(parse-error? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(unexpected-eof-from-server? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(bad-address? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(no-nameservers? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(bad-nameserver? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(not-a-hostname? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(not-a-ip? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(dns-format-error? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(dns-server-failure? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(dns-name-error? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(dns-not-implemented? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure
(dns-refused? <i>thing</i>) \longrightarrow <i>boolean</i>	procedure

The type predicates for the conditions above.

13.3 High-level Interface

The library uses an internal store to cache data obtained from DNS servers. All procedures take a boolean flag *use-cache?* that indicates whether the cache should be used or not. *use-cache?* defaults to true.

`(dns-clear-cache!) → undefined` procedure

This procedure erases all information stored in the internal cache.

The library is further capable of parsing the contents of `/etc/resolv.conf` (see Section 13.5). The nameservers listed there are the default value for the optional argument *nameserver list* which many procedures of the library accept. *Nameserver* is either a IP-address or a dotted IP string.

`(dns-lookup-name FQDN [nameserver list][use-cache?]) → IP-address` procedure

Given the FQDN of a host, `dns-lookup-ip` returns the IP address. The optional argument specifies the name servers to query, it defaults to the ones found in `/etc/resolv.conf`.

`(dns-lookup-ip IP-string/IP-address [nameserver list][use-cache?]) → FQDN` procedure

Looks up the FQDN for the given IP address. The optional argument specifies the name servers to query, it defaults to the ones found in `/etc/resolv.conf`.

`(dns-lookup-nameserver IP-string/IP-address [nameserver list][use-cache?]) → IP-address list` procedure

Looks up an authoritative name server for a hostname, returns a list of name servers.

`(dns-lookup-mail-exchanger IP-string/IP-address [nameserver list][use-cache?]) → FQDN list` procedure

Looks up mail-exchangers for a hostname und returns them in a list sorted by preference.

`(socket-address->fqdn socket-address [nameserver list][use-cache?]) → FQDN` procedure

Returns the FQDN for of the address bound to argument. The argument *cache?* indicates whether the internal cache may be queried to obtain the information.

`(maybe-dns-lookup-name FQDN [nameserver list][use-cache?]) → IP-address or #f` procedure

`(maybe-dns-lookup-ip IP-string/IP-address [nameserver list][use-cache?]) → FQDN or #f` procedure

These procedures provide the same functionality as `dns-lookup-name` and `dns-lookup-ip` but return `#f` in case of an `dns-error`.

(*host-fqdn name/socket-address [nameserver list][use-cache?]*) → *FQDN* procedure
 (*system-fqdn [nameserver list][use-cache?]*) → *FQDN* procedure

host-fqdn returns the fully qualified domain name (FQDN) for its argument which can be either a unqualified host name or a socket address. The procedure *system-fqdn* returns the FQDN of the local host. These procedures use a list of domain names obtained from */etc/resolv.conf* to generate FQDNs and try to resolve these FQDNs.

13.4 Low-level Interface

This section describes a set of data structures and procedures which directly correspond to the data flow of the DNS protocol. The central entity is a *message*, the abstraction of the packet sent to the server or received from the server (The DNS protocol uses the same data format for both directions). A *dns-message* encapsulates the query message sent to the server, the response message received from the server, and some additional information the library gathered while generating the *dns-message*.

(*dns-get-information message protocol answer-okay? [nameserver list][use-cache?]*) → *dns-message* procedure

Most general way to submit a DNS query. The message is sent to the name servers via *protocol* which can be either (*network-protocol tcp*) or (*network-protocol udp*), both members of the enumerated type *network-protocol*. After receiving the reply, *dns-get-information* applies the predicate *answer-okay?* to the message. If it returns *#f* and the answer is not authoritative additional name servers sent with the reply are checked until an authoritative answer is found. If the predicate returns *#f* but the answer is authoritative a *bad-address* condition is signalled.

(*network-protocol protocol-name*) → *network-protocol* syntax
 (*network-protocol? thing*) → *boolean* procedure

Constructor and predicate for the enumerated type *network-protocol* with the possible protocol names *tcp* and *udp*.

(*dns-lookup IP-string/IP-address type [nameserver list][use-cache?]*) → *dns-message* procedure

Convenient shortcut to submit a DNS query. The return value is a *dns-message* structure:

(*dns-message? thing*) → *boolean* procedure
 (*dns-message-query dns-message*) → *message* procedure
 (*dns-message-reply dns-message*) → *message* procedure
 (*dns-message-cache? dns-message*) → *boolean* procedure
 (*dns-message-protocol dns-message*) → *protocol* procedure
 (*dns-message-tried-nameservers dns-message*) → procedure

A *dns-message* records the query sent to the server and the reply from the server. It also contains information whether the library took the reply from the cache, which protocol was used and to which nameservers the query was sent.

(pretty-print-dns-message *dns-message* [*output-port*]) \rightarrow *undefined* procedure

Pretty prints a DNS message to *out-port* which defaults to the current output port.

(message? <i>thing</i>)	\rightarrow <i>boolean</i>	procedure
(message-header <i>message</i>)	\rightarrow <i>header</i>	procedure
(message-questions <i>message</i>)	\rightarrow <i>question list</i>	procedure
(message-answers <i>message</i>)	\rightarrow <i>resource-record list</i>	procedure
(message-nameservers <i>message</i>)	\rightarrow <i>resource-record list</i>	procedure
(message-additionals <i>message</i>)	\rightarrow <i>resource-record list</i>	procedure
(message-source <i>message</i>)	\rightarrow <i>char list</i>	procedure

A message represents the data sent to the DNS server or received from the DNS server. The DNS protocol uses the same message format for queries and replies. In queries only the header and the questions is present, a reply may contain answers, name servers and and additional informations as resource records. Message-source returns the actual data sent over the network.

(make-query-message *header header question* [*questions*]) \rightarrow *message* procedure

The procedure generates a message the supplied questions, *header*, and the standard message values for queries.

(make-simple-query-message *name type class*) \rightarrow *message* procedure

This simplified constructor generates a message with one question which is built from the parameters, and the standard header flags for queries and the standard message values for queries.

(header? <i>thing</i>)	\rightarrow <i>boolean</i>	procedure
(header-id <i>header</i>)	\rightarrow <i>number</i>	procedure
(header-flags <i>header</i>)	\rightarrow <i>flags</i>	procedure
(header-question-count <i>header</i>)	\rightarrow <i>number</i>	procedure
(header-answer-count <i>header</i>)	\rightarrow <i>number</i>	procedure
(header-nameserver-count <i>header</i>)	\rightarrow <i>number</i>	procedure
(header-additional-count <i>header</i>)	\rightarrow <i>number</i>	procedure

Every DNS message contains a header which stores information about the data present in the message and contains flags for the query.

(flags? <i>thing</i>)	→ <i>boolean</i>	procedure
(flags-query-type <i>flags</i>)	→ <i>'query or 'response</i>	procedure
(flags-opcode <i>flags</i>)	→ <i>number</i>	procedure
(flags-authoritative? <i>flags</i>)	→ <i>boolean</i>	procedure
(flags-truncated? <i>flags</i>)	→ <i>boolean</i>	procedure
(flags-recursion-desired? <i>flags</i>)	→ <i>boolean</i>	procedure
(flags-recursion-available? <i>flags</i>)	→ <i>boolean</i>	procedure
(flags-z <i>flags</i>)	→ <i>0</i>	procedure
(flags-response-code <i>flags</i>)	→ <i>number</i>	procedure

Flags occur within the header of a DNS message. The boolean value returned from flags-authoritative indicates whether the message was sent from a authoritative server, flags-truncated? should always be #f as the library automatically uses the TCP protocol is the UDP message size is not sufficed.

(question? <i>thing</i>)	→ <i>boolean</i>	procedure
(question-name <i>question</i>)	→ <i>string</i>	procedure
(question-type <i>question</i>)	→ <i>message-type</i>	procedure
(question-class <i>question</i>)	→ <i>message-class</i>	procedure

A question sent to the DNS server.

The type and class of the question and answer are elements of enumerated types:

(message-class <i>class-name</i>)	→ <i>message-class</i>	syntax
(message-class? <i>thing</i>)	→ <i>boolean</i>	procedure
(message-class-name <i>message-class</i>)	→ <i>symbol</i>	procedure
(message-class-number <i>message-class</i>)	→ <i>number</i>	procedure

message-class constructs a member of the enumerated type, message-class? is the type predicate, message-class-name returns the symbol and message-class-number the number used for the class in the DNS protocol.

The possible names for the classes are:

in The Internet

cs obsolete

ch the CHAOS class

hs Hesoid

(message-type <i>type-name</i>)	→ <i>message-type</i>	syntax
(message-type? <i>thing</i>)	→ <i>boolean</i>	procedure

`(message-type-name message-type)` \rightarrow *symbol* procedure
`(message-type-index message-type)` \rightarrow *number* procedure

`message-type` constructs a member of the enumeration from name $\langle type-name \rangle$ listed in Table 13.1. `message-type?` is the type predicate, `message-type-name` returns the name, and `message-type-number` the number used for the class the DNS protocol.

<code>a</code>	a host address
<code>ns</code>	an authoritative name server
<code>md</code>	(obsolete)
<code>mf</code>	(obsolete)
<code>cname</code>	the canonical name for an alias
<code>soa</code>	marks the start of a zone of authority
<code>mb</code>	(experimental)
<code>mg</code>	(experimental)
<code>mr</code>	(experimental)
<code>null</code>	(experimental)
<code>wks</code>	a well known service description
<code>ptr</code>	a domain name pointer
<code>hinfo</code>	host information
<code>minfo</code>	(experimental)
<code>mx</code>	mail exchange
<code>txt</code>	text strings

Table 13.1: Message types

`(resource-record? thing)` \rightarrow *boolean* procedure
`(resource-record-name resource-record)` \rightarrow *string* procedure
`(resource-record-type resource-record)` \rightarrow *message-type* procedure
`(resource-record-class resource-record)` \rightarrow *message-class* procedure
`(resource-record-ttl resource-record)` \rightarrow *number* procedure
`(resource-record-data resource-record)` \rightarrow *resource-record-data-...* procedure

A resource record as returned from the DNS server. The actual data of the record is stored in the `resource-record-data` field. It is one of the record types for resource record data described below.

`(resource-record-data-a? thing)` \rightarrow *boolean* procedure
`(resource-record-data-a-ip resource-record-data-a)` \rightarrow *IP-address* procedure

An address resource record which holds an internet address.

`(resource-record-data-ns? thing)` \rightarrow *boolean* procedure

(resource-record-data-ns-name *resource-record-data-ns*) \rightarrow FQDN procedure

A name server resource record containing the FQDN of the name server.

(resource-record-data-cname? *thing*) \rightarrow boolean procedure
(resource-record-data-cname *resource-record-data-cname*) \rightarrow FQDN procedure

A canonical name resource record which contains the canonical or primary name of the owner.

(resource-record-data-mx? *thing*) \rightarrow boolean procedure
(resource-record-data-mx-preference *resource-record-data-mx*) \rightarrow number procedure
(resource-record-data-mx-exchanger *resource-record-data-mx*) \rightarrow FQDN procedure

A mail exchange resource record with the preference and the FQDN of a host willing to act as a mail exchange.

(resource-record-data-ptr? *thing*) \rightarrow boolean procedure
(resource-record-data-ptr-name *resource-record-data-ptr*) \rightarrow string procedure

A pointer resource record which points to some other domain name.

(resource-record-data-soa? *thing*) \rightarrow boolean procedure
(resource-record-data-soa-mname *resource-record-data-soa*) \rightarrow FQDN procedure
(resource-record-data-soa-rname *resource-record-data-soa*) \rightarrow FQDN procedure
(resource-record-data-soa-serial *resource-record-data-soa*) \rightarrow number procedure
(resource-record-data-soa-refresh *resource-record-data-soa*) \rightarrow number procedure
(resource-record-data-soa-retry *resource-record-data-soa*) \rightarrow number procedure
(resource-record-data-soa-expire *resource-record-data-soa*) \rightarrow number procedure
(resource-record-data-soa-minimum *resource-record-data-soa*) \rightarrow number procedure

A start of a zone of authority resource record.

The protocol specifies other possible values for the resource-record-data field but we were not able to find test cases for them.

(cache? *thing*) \rightarrow boolean procedure
(cache-answer *cache*) \rightarrow dns-message procedure
(cache-ttl *cache*) \rightarrow number procedure
(cache-time *cache*) \rightarrow number procedure

A cache data structure corresponds to a saved answer to a previous query. cache-answer returns the saved message, cache-ttl returns the time when the cache entry expires and cache-time returns the time the entry was created.

13.5 Parsing /etc/resolv.conf

`resolv.conf-parse-error` condition

`(resolv.conf-parse-error? thing) → boolean` procedure

The code signals the condition *resolv.conf-parse-error* if a parse error occurs while scanning */etc/resolv.conf*. It is a subtype of the *dns-error* condition. `resolv.conf-parse-error?` is the type predicate for this condition.

`(resolv.conf) → symbol→string alist` procedure

Returns the contents of */etc/resolv.conf* as an alist with the possible keys *nameserver*, *domain*, *search*, *sortlist* and *options*.

Note that the library caches the contents of */etc/resolv.conf* and *resolv.conf* only really opens the file if its modification time is more recent than the modification time of the cache.

`(parse-resolv.conf!) → undefined` procedure

Parses the contents of */etc/resolv.conf* and updates the internal cache of the library.

`(dns-find-nameserver-list) → FQDN list` procedure

Returns a list of name servers from */etc/resolv.conf*

`(dns-find-nameserver) → FQDN` procedure

Returns the first name servers found in */etc/resolv.conf*. *dns-find-nameserver* raises *no-nameservers* if */etc/resolv.conf* does not contain a *nameserver* entry.

`(domains-for-search) → string list` procedure

Parses */etc/resolv.conf* and extracts the domains specified by the search keyword.

13.6 IP Addresses as Dotted Strings

Used files: *ip.scm*

Name of the package: *ips*

The structure `ips` provides a small set of procedures for turning the human-readable form of IP addresses (“dotted strings”) into 32 bits numbers.

<code>(address32->ip-string IP-address)</code>	\longrightarrow	<code>ip-string</code>	procedure
<code>(ip-string->address32 ip-string)</code>	\longrightarrow	<code>IP-address</code>	procedure
<code>(ip-string? string)</code>	\longrightarrow	<code>boolean</code>	procedure

Tests whether *string* is a valid dotted string for an IP address.

Index

address32->ip-string, 54
alist-path-dispatcher, 13

bad-address, 46
bad-address?, 46
bad-nameserver, 46
bad-nameserver?, 46

cache-answer, 52
cache-time, 52
cache-ttl, 52
cache?, 52
cgi-form-query, 27
cgi-handler, 17
copy-ascii-port->port, 33
copy-port->port-ascii, 33
copy-port->port-binary, 33

dns-error, 45
dns-error->string, 46
dns-error?, 45
dns-find-nameserver, 53
dns-find-nameserver-list, 53
dns-format-error, 46
dns-format-error?, 46
dns-get-information, 48
dns-lookup, 48
dns-lookup-ip, 47
dns-lookup-mail-exchanger, 47
dns-lookup-name, 47
dns-lookup-nameserver, 47
dns-message-cache?, 48
dns-message-protocol, 48
dns-message-query, 48
dns-message-reply, 48
dns-message-tried-nameservers, 48
dns-message?, 48
dns-name-error, 46
dns-name-error?, 46
dns-not-implemented, 46
dns-not-implemented?, 46
dns-refused, 46
dns-refused?, 46
dns-server-error, 46
dns-server-error?, 46
dns-server-failure, 46
dns-server-failure?, 46
domains-for-search, 53

escape-uri, 21
eval-safely, 18

flags-authoritative?, 50
flags-opcode, 50
flags-query-type, 50
flags-recursion-available?, 50
flags-recursion-desired?, 50
flags-response-code, 50
flags-truncated?, 50
flags-z, 50
flags?, 50
ftp-append, 32
ftp-cd, 32
ftp-cdup, 32
ftp-connect, 31
ftp-delete, 32
ftp-dir, 32

- ftp-error?, 33
- ftp-get, 32
- ftp-inetd, 28
- ftp-ls, 32
- ftp-mkdir, 33
- ftp-modification-time, 33
- ftp-put, 32
- ftp-pwd, 32
- ftp-quit, 33
- ftp-quot, 33
- ftp-rename, 31
- ftp-rmdir, 32
- ftp-size, 33
- ftp-type, 31
- ftpd, 28

- header-additional-count, 49
- header-answer-count, 49
- header-flags, 49
- header-id, 49
- header-nameserver-count, 49
- header-question-count, 49
- header?, 49
- home-dir-handler, 16
- host-fqdn, 48
- http-url->string, 25
- http-url-fragment-identifier, 25
- http-url-path, 25
- http-url-search, 25
- http-url-server, 25
- http-url?, 25
- httpd, 7

- ip-string->address32, 54
- ip-string?, 54

- loser, 18

- make-error-response, 10
- make-file-directory-options, 16
- make-ftpd-options, 29
- make-host-name-handler, 13
- make-http-url, 25
- make-httpd-options, 9
- make-path-predicate-handler, 13
- make-path-prefix-handler, 13
- make-predicate-handler, 13
- make-query-message header, 49
- make-reader-writer-body, 12
- make-redirect-response, 10
- make-response, 10
- make-server, 24
- make-simple-query-message, 49
- make-writer-body, 12
- maybe-dns-lookup-ip, 47
- maybe-dns-lookup-name, 47
- message-additionals, 49
- message-answers, 49
- message-class, 50
- message-class-name, 50
- message-class-number, 50
- message-class?, 50
- message-header, 49
- message-nameservers, 49
- message-questions, 49
- message-source, 49
- message-type, 50
- message-type-index, 51
- message-type-name, 51
- message-type?, 50
- message?, 49

- name->status-code, 11
- netrc-entry-account, 35
- netrc-entry-login, 35
- netrc-entry-machine, 35
- netrc-entry-password, 35
- netrc-entry?, 35
- netrc-machine-entry, 34
- netrc-macro-definitions, 35
- network-protocol, 48
- network-protocol?, 48
- no-nameservers, 46
- no-nameservers?, 46
- not-a-hostname, 46
- not-a-hostname?, 46
- not-a-ip, 46
- not-a-ip?, 46

null-request-handler, 13	resource-record-data-a-ip, 51
	resource-record-data-a?, 51
parse-error, 46	resource-record-data-cname, 52
parse-error?, 46	resource-record-data-cname?, 52
parse-html-form-query, 19	resource-record-data-mx-exchanger,
parse-http-url, 25	52
parse-http-url-string, 26	resource-record-data-mx-preference,
parse-server, 24	52
parse-uri, 20	resource-record-data-mx?, 52
pop3-connect, 43	resource-record-data-ns-name, 52
pop3-delete, 44	resource-record-data-ns?, 51
pop3-error?, 44	resource-record-data-ptr-name, 52
pop3-last, 44	resource-record-data-ptr?, 52
pop3-quit, 44	resource-record-data-soa-expire,
pop3-reset, 44	52
pop3-retrieve-headers, 44	resource-record-data-soa-minimum,
pop3-retrieve-message, 44	52
pop3-stat, 44	resource-record-data-soa-mname,
pretty-print-dns-message, 49	52
	resource-record-data-soa-refresh,
	52
question-class, 50	resource-record-data-soa-retry,
question-name, 50	52
question-type, 50	resource-record-data-soa-rname,
question?, 50	52
	resource-record-data-soa-serial,
read-rfc822-field, 36	52
read-rfc822-field-with-line-breaks,	resource-record-data-soa?, 52
36	resource-record-name, 51
read-rfc822-headers, 37	resource-record-ttl, 51
read-rfc822-headers-with-line-breaks,	resource-record-type, 51
37	resource-record?, 51
request-handler, 12	rfc822-time->string, 37
request-headers, 10	rfc867-daytime/tcp, 38
request-method, 9	rfc867-daytime/udp, 38
request-socket, 10	rfc868-time/tcp, 39
request-uri, 9	rfc868-time/udp, 39
request-url, 9	rooted-file-handler, 16
request-version, 9	rooted-file-or-directory-handler,
request?, 9	16
resolv.conf, 53	
resolv.conf-parse-error, 53	server->string, 24
resolv.conf-parse-error?, 53	server-host, 24
resource-record-class, 51	server-password, 24
resource-record-data, 51	

- server-port, 24
- server-user, 24
- server?, 24
- seval-handler, 17
- simplify-uri-path, 22
- smtp-connect, 41
- smtp-data, 42
- smtp-error?, 40
- smtp-expand, 41
- smtp-expn, 42
- smtp-get-help, 41
- smtp-helo, 42
- smtp-help, 42
- smtp-mail, 42
- smtp-noop, 42
- smtp-quit, 42
- smtp-rcpt, 42
- smtp-recipients-rejected-error?, 40
- smtp-rset, 42
- smtp-saml, 42
- smtp-send, 42
- smtp-send-mail, 40
- smtp-soml, 42
- smtp-transactions, 41
- smtp-transactions/no-close, 41
- smtp-turn, 42
- smtp-verify, 41
- smtp-vrfy, 42
- socket-address->fqdn, 47
- split-uri, 22
- status-code, 11
- status-code-message, 11
- status-code-number, 11
- system-fqdn, 48
- tilde-home-dir-handler, 16
- unescape-uri, 21
- unexpected-eof-from-server, 46
- unexpected-eof-from-server?, 46
- uri-escaped-chars, 21
- uri-path->uri, 22
- with-anonymous-home, 28
- with-back-icon-url, 15
- with-banner, 28
- with-blank-icon-url, 15
- with-dns-lookup?, 29
- with-file-name->content-encoding, 15
- with-file-name->content-type, 15
- with-file-name->icon-url, 15
- with-fqdn, 8
- with-log-file, 8
- with-log-port, 28
- with-port, 8, 28
- with-reported-port, 8
- with-request-handler, 8
- with-resolve-ip?, 9
- with-root-directory, 8
- with-server-admin, 8
- with-simultaneous-requests, 8
- with-syslog?, 9
- with-unknown-icon-url, 16