

# **SUnet Reference Manual**

---

For SUnet release 2.2  
October 2006

**Dr. S<sup>2</sup>, Martin Gasbichler, Eric Marsden, Andreas Bernauer**

---

# Contents

<b>Contents</b>	<b>2</b>
<b>1 Overview</b>	<b>6</b>
1.1 Obtaining the system . . . . .	7
1.2 How to install SUnet . . . . .	7
1.3 How to use the packages . . . . .	7
<b>2 HTTP server</b>	<b>9</b>
2.1 Starting and configuring the server . . . . .	9
2.2 Requests . . . . .	11
2.3 Responses . . . . .	12
2.4 Response Bodies . . . . .	14
2.5 Request Handlers . . . . .	14
2.5.1 Basic Request Handlers . . . . .	15
2.5.2 Static Content Request Handlers . . . . .	16
2.6 CGI Server . . . . .	19
2.7 Scheme-Evaluating Request Handlers . . . . .	19
2.7.1 The loser structure . . . . .	20
2.7.2 The toothless structure . . . . .	20
2.7.3 The toothless-eval structure . . . . .	20
2.8 Writing Request Handlers . . . . .	21
2.8.1 Parsing HTML Forms . . . . .	21
2.9 SSL encryption with Apache . . . . .	21

<b>3</b>	<b>Parsing and Processing URIs</b>	<b>23</b>
3.1	Notes on URI Syntax . . . . .	23
3.2	Procedures . . . . .	23
<b>4</b>	<b>Parsing and Processing URLs</b>	<b>27</b>
4.1	Server Records . . . . .	27
4.2	HTTP URLs . . . . .	28
<b>5</b>	<b>Writing CGI Scripts in Scheme</b>	<b>30</b>
<b>6</b>	<b>SUrflet server</b>	<b>31</b>
6.1	Howto . . . . .	31
6.1.1	Introduction . . . . .	31
6.1.2	How to run the SUnet webserver that handles SUrflets .	32
6.1.3	How to send web pages . . . . .	34
6.1.4	How to write web forms . . . . .	39
6.1.5	Program flow control . . . . .	48
6.1.6	Data management . . . . .	53
6.1.7	My own SXML . . . . .	55
6.2	API description . . . . .	57
6.2.1	The SUrflet server . . . . .	57
6.2.2	SUrflets . . . . .	59
6.2.3	SUrflet management . . . . .	60
6.2.4	Surflet Request . . . . .	61
6.2.5	Surflet Response . . . . .	61
6.2.6	Sessions . . . . .	62
6.2.7	Basic I/O . . . . .	66
6.2.8	Web I/O . . . . .	67
6.2.9	Continuation-URL . . . . .	71
6.2.10	Input fields . . . . .	72
6.2.11	Web addresses . . . . .	80
6.2.12	Callbacks . . . . .	81
6.2.13	Outdater . . . . .	83
6.2.14	Simple SUrflets . . . . .	84
<b>7</b>	<b>FTP Server</b>	<b>87</b>

<b>8</b>	<b>FTP Client</b>	<b>90</b>
<b>9</b>	<b>Parsing Netrc Files</b>	<b>93</b>
<b>10</b>	<b>RFC 822 Library</b>	<b>95</b>
<b>11</b>	<b>Time and Daytime</b>	<b>97</b>
11.1	Daytime . . . . .	97
11.2	Time . . . . .	97
<b>12</b>	<b>SMTP Client</b>	<b>99</b>
<b>13</b>	<b>POP3 Client</b>	<b>102</b>
<b>14</b>	<b>DNS Client Library</b>	<b>104</b>
14.1	Overview . . . . .	104
14.2	Conditions . . . . .	104
14.3	High-level Interface . . . . .	106
14.4	Low-level Interface . . . . .	107
14.5	Parsing <code>/etc/resolv.conf</code> . . . . .	112
14.6	IP Addresses as Dotted Strings . . . . .	112
	<b>Index</b>	<b>114</b>

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 server also ships with a sophisticated interface for writing server-side Web applications called *SUrflets*.

**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.scsch.net/resources/sunet.html>. To run the code, you need version 0.6.6 or later of scsh from <http://www.scsch.net/>.

## 1.2 How to install SUnet

Starting with version 2.1 SUnet conforms to the packaging proposal for scsh by Michel Schinz and needs Michel's installation library to install properly. For more information, please see [http://lamp.epfl.ch/~schinz/scsh\\_packages/](http://lamp.epfl.ch/~schinz/scsh_packages/).

In short, this means that you can install SUnet by unpacking the SUnet tarball and issuing the following command in the created directory:

```
scsh-install-pkg --prefix /path/to/your/package/root
```

See the file INSTALL for the generic installation instructions for scsh packages.

You need to install version 4.9 of the SSAX package to use SUnet. SSAX is available from [http://lamp.epfl.ch/~schinz/scsh\\_packages/](http://lamp.epfl.ch/~schinz/scsh_packages/).

## 1.3 How to use the packages

After installation, you can use the `-lel` command-line option to load the package definitions. If you installed SUnet including SURflets (the default), you need to load SSAX as well:

```
atari-2600[72] scsh -lel SSAX-4.9/load.scm -lel sunet-2.1/load.scm
Welcome to scsh 0.6.6 (King Conan)
Type ,? for help.
```

Now, all structures defined by SUnet and SSAX are available:

```
> ,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-ips? 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 socket connected to the client.<sup>1</sup>

## 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

---

<sup>1</sup>Request 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 &lt;name&gt;)</code>	$\longrightarrow$	<i>status-code</i>	syntax
<code>(name-&gt;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) → 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) → 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) → 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/1.1<sup>2</sup>.

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.

---

<sup>2</sup>see <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) → 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.

## 2.9 SSL encryption with Apache

Network traffic with a HTTP server is usually encrypted and protected from manipulation using the cryptographic algorithm provided by an implementation of the *secure socket layer*, SSL for short. SUnet does not have support for SSL yet. However, an Apache web-server with SSL support can be configured as a proxy. In this setup the Apache web-server accepts encrypted requests and forwards them to a SUnet web-server running locally. This section describes how to set up Apache as an encrypting proxy, assuming the reader has basic knowledge about Apache and its configuration directives.

The following excerpt shows a minimalist SSL virtual host that forwards requests to a SUnet server.

```
<VirtualHost 134.2.12.82:443>
  DocumentRoot "/www/some-domain/htdocs"
  ServerName www.some-domain.de
  ServerAdmin admin@some-domain.de
  ErrorLog /www/some-domain/logs/error_log

  ProxyRequests off
  ProxyPass / http://localhost:8080/
  ProxyPassReverse / http://localhost:8080/

  SSLEngine on
  SSLRequireSSL
```

```
SSLCertificateFile /www/some-domain/cert/some-domain.cert
SSLCertificateKeyFile /www/some-domain/cert/some-domain.key
</VirtualHost>
```

First, a virtual host is added to Apache's configuration file. This virtual host listens for incoming connections on port 443, which is the standard port for encrypted HTTP traffic. `SSLRequireSSL` ensures that server accepts encrypted connections only.

In terms of the Apache documentation, the web-server acts as a so called *reverse proxy*. The option `ProxyRequests` has a misleading name. Setting this option to off does only turns off Apache's facility to act as a *forward proxy* and has no effect on the configuration directives for reverse proxies. Actually, turning on `ProxyRequests` is dangerous, because this turns Apache into a proxy server that can be used from anywhere to access any site that is accessible to the Apache server.

In this setting, all requests get forwarded to a SUnet web-server which listens for incoming connections on localhost port 8080 only, thus, it is not reachable from a remote machine. Apache forwards all requests to the host and port specified by the `ProxyPass` directive. `ProxyPassReverse` specifies how Location-Header fields of HTTP redirect messages send by the SUnet server are translated.

## 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) → 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 Net<sup>1</sup>, e.g. = is a reserved character, but used unescaped in the search-part. It was given to me<sup>2</sup> by Dan Connolly of the W3C and slightly modified.

(unescape-uri *string* [*start*] [*end*])  $\longrightarrow$  *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. RFC 2396 defines this set as all characters which are neither letters, nor digits, nor one of the following characters: -, ., !, ~, \*, ', (, ).

(escape-uri *string* [*escaped-chars*])  $\longrightarrow$  *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

<sup>1</sup>So it does not absolutely conform to RFC 1630.

<sup>2</sup>That'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

The `url` structure 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.

<code>(make-server user password host port)</code>	$\longrightarrow$ <i>server</i>	procedure
<code>(server? thing)</code>	$\longrightarrow$ <i>boolean</i>	procedure
<code>(server-user server)</code>	$\longrightarrow$ <i>string-or-#f</i>	procedure
<code>(server-password server)</code>	$\longrightarrow$ <i>string-or-#f</i>	procedure
<code>(server-host server)</code>	$\longrightarrow$ <i>string-or-#f</i>	procedure
<code>(server-port server)</code>	$\longrightarrow$ <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.

<code>(parse-server path default)</code>	$\longrightarrow$ <i>server</i>	procedure
<code>(server-&gt;string server)</code>	$\longrightarrow$ <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.scsn.net" "docu" "index.html")
               default)
'#server
> (server->string ##)
"foo%20bar:se%20ret@www.scsn.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->uri (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-&gt;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

# SURflet server

The SURflet server enables you to write server side scripted web programs in Scheme. There are lots of example files in `scheme/httpd/surflet/webserver/root/surflets` from which you can copy freely.

### 6.1 Howto

This howto gives a short introduction in how to write a SURflet. It is concentrated on the practical side rather on describing the SURflet API in detail to give you instant succes in running your own surflets. See section 6.2 for the (technical) API description.

#### 6.1.1 Introduction

For those who don't know it already, SURflets are pieces of code that can be executed interactively through a website. There is a SURflet handler who administrates their execution and suspension. The SURflet handler is part of the SUNet webserver. SURflets ease the implementation of web applications in two ways, compared to other server-side scripting tools like Java<sup>TM</sup>Servlets or Microsoft® Active Server Pages or PHP:

1. SURflets have an automatic program flow control like any other usual program (but unlike usual web programs), *i.e.* the web designer doesn't have to care about session management at all. The sequence of the web pages result from their appearance in the program like the print statements in any other usual program.

2. SURflets come along with a library for robust user interaction. SURflets represent interaction elements of the web page like text input fields or dropdown lists in the SURflet program by specific objects. A web designer can plug in these objects into a website and use them to read out the user input.

The following sections probably assume that you have basic knowledge of the SUNet webserver and scsh. The environment variable `$sunet` refers to the top level directory of your sunet installation. On my system this is `/home/andreas/sw/sunet`.

### 6.1.2 How to run the SUNet webserver that handles SURflets

The following sections will show pieces of SURflet code you might want to try out. Therefore you need the SUNet webserver running with the ability to serve SURflets. This section tells you how to do it.

**Obtaining necessary packages** You need Oleg's SSAX package (for scsh), to be able to use surflets:

- Download Oleg's SSAX package from <http://prdownloads.sf.net/ssax/ssax-sr5rs-plt200-4.9.tar.gz?download>.
- Download the SSAX package kit from [http://lamp.epfl.ch/~schinz/scsh\\_packages](http://lamp.epfl.ch/~schinz/scsh_packages).
- Uncompress and untar both tarballs in the same directory. This will create a directory called SSAX, to which I will refer to as `$$SSAX`. The package kit will add a file `pkg-def.scm` to the SSAX directory.
- Install SSAX as a scsh package by issuing the command `scsh-install-pkg --prefix /path/to/your/package/root` in the `$$SSAX` directory. If you don't yet have the packaging utility of Michel Schinz, you can obtain it from [http://lamp.epfl.ch/~schinz/scsh\\_packages](http://lamp.epfl.ch/~schinz/scsh_packages).

If you don't want to install SSAX with the packaging utility, you can adjust the scripts to directly load the SSAX package definitions from `$$SSAX/lib/packages.scm`. *Note that the original file has a typo which you can correct with*

```
cd $$SSAX
patch -p1 < $sunet/httpd/surflets/SSAX-goodhtml-patch
```



**Starting the SURflet server** You can start the SUnet webserver along with the SURflet-handler now. The SUnet distribution comes with a script that does this for you:

```
$ /path/to/your/package/root/sunet/web-server/start-surflet-server
```

Please be patient, scsh has to load a lot of libraries. If the loading succeeds you will see something like this:

```
[andreas@hgt web-server]$ ./start-surflet-server
Loading...
reading options: ()
Going to run SURflet server with:
htdocs-dir:      /home/andreas/bin/lib/scsh/0.6/sunet-2.1/web-server/root/htdocs
surflet-dir:     /home/andreas/bin/lib/scsh/0.6/sunet-2.1/web-server/root/surflets
images-dir:     /home/andreas/bin/lib/scsh/0.6/sunet-2.1/web-server/root/img
port:           8080
log-file-name:  /tmp/httpd.log
a maximum of 5 simultaneous requests, syslogging activated,
and home-dir-handler (public_html) activated.
```

NOTE: This is the SURflet server. It does not support cgi.

This means the server is up and running. Try to connect to `http://localhost:8080` with your browser and you will see the welcome page of the SUnet server. There's a link to the SURflets homepage. You can also already try out some of the SURflets that come with the distribution.

You will probably notice a long response time the first time you load the first SURflet. This is because the server has to load the SURflet libraries. The server handles further requests to SURflets faster.

If the port the SURflet server tries to use is occupied, you will see an error message similar to this one:

```
Error: 98
      "Address already in use"
      #Procedure 11701 (%bind in scsh-level-0)
      4
      2
      (0 . 8080)
```

In this case, pass another port number to the script, *e.g.* 8000:

```
start-surflet-server -p 8000
```

The `--help` option will show you more parameters that you can adjust, but you won't need them for this howto.

### 6.1.3 How to send web pages

This section will discuss some of the various ways in which you can send a web page to a browser that contacted your SURflet.

#### My first SURflet

Traditionally, your first program in any programming language prints something like “Hello, World!”. We follow this tradition:

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scsh)
  (begin
    (define (main req)
      (send-html/finish
        '(html (body (h1 "Hello, world!")))))
  ))
```

You can either save a file with that content in the SURflets directory the server mentioned at startup or you can use the file `howto/hello.scm` that comes along with the SURflets distribution and which is located in the server’s standard SURflets directory. Let’s go through the small script step by step:

```
(define-structure surflet surflet-interface
```

This defines a module named `surflet` which implements the interface `surflet-interface`. `surflet-interface` just states that the module exports a function named `main` to which we will come shortly. For those of you who know about the `scsh` module system: Yes, SURflets are basically `scsh` modules that are loaded dynamically during run time.

```
  (open surflets
    scheme-with-scsh)
```

The `open` form lists all the modules the SURflet needs. You will probably always need the two modules that are stated here (namely `surflets` and `scheme-with-scsh`). If you need other modules, like `srfi-13` for string manipulation, this is the place where you want to state it.

```
  (begin
```

This just opens the body of the SURflet. All your SURflet code goes here.<sup>1</sup>

---

<sup>1</sup>If you know about `scsh` modules, you probably also know that there is a `file` clause that you could use to place the code in a file instead or along with the `begin` clause.

```
(define (main req)
```

Here is the main function that the interface declared this SURflet will implement. The main function is the entry point to your SURflet: The server calls this function every time a user browses to your SURflet the first time. The server calls main with one argument: a representation of the initial request of the browser. We don't have to worry about that at this point.

```
(send-html/finish
  '(html (body (h1 "Hello, world!")))))
))
```

`send-html/finish` is one of three functions you will regularly use to send web pages to the browser. The other two functions are `send-html` and `send-html/suspend`. `send-html/finish` – as the name already suggests – sends a HTML page to the browser and finishes the SURflet. `send-html` just sends the HTML page and does not return. `send-html/suspend` sends the HTML page and suspends the SURflet, *i.e.* it waits until the user continues with the SURflet, *e.g.* by submitting a webform. We will discuss `send-html` and `send-html/suspend` in detail later. I will refer to these three functions as the *sending functions*.

In a SURflet, HTML pages are represented as lists, or, to be more precise, as SXML (S-expression based XML). The first element of a SXML list is a symbol stating the HTML tag. The other elements of a SXML list are the contents that are enclosed by this HTML tag. The contents can be other SXML lists, too. Here are some examples of SXML lists and how they translate to HTML:

```
SXML:  '(p "A paragraph.")}
HTML:  <p>A paragraph.\htmltag{/p>}
```

```
SXML:  '(p "A paragraph." (br) "With break line.")}
HTML:  <p>A paragraph.<br>With break line.</p>}
```

```
SXML:  '(p "Nested" (p "paragraphs"))}
HTML:  <p>Nested<p>paragraphs</p></p>}
```

Attributes are stated by a special list whose first element is the `at`-symbol. The attribute list must be the second element in the list:

```
SXML:  '(a (@ (href "attr.html")) "Attributed HTML tags.")
HTML:  <a href="attr.html">Attributed HTML tags.</a>
```

```
SXML:  '(a (@ (href "attr2.html") (target "_blank")) "2 attributes.")}
HTML:  <a href="attr2.html" target="_blank">2 attributes.</a>}
```

As you see from the SURflet example, `send-html/finish` expects SXML as an argument. In the example, the SXML translates to the following HTML code:

```
<html><body><h1>Hello, world!</h1>
</body>
</html>
```

Please note, that there is no check for valid HTML or even XHTML here. The only thing the translation process takes care of are special characters in strings like the ampersand (&). The translation process replaces them by their HTML representation (*e.g.*, &amp;) so you don't have to worry about that when you use strings. Everything else like using valid HTML tags or valid attributes is your responsibility.

## Dynamic content

Let's extend our first SURflet example by some dynamic content, *e.g.* by displaying the current time using `scsh`'s `format-date` function. As the HTML page is basically represented as a list, this can be done like this:

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scsh)
  (begin
    (define (main req)
      (send-html/finish
        '(html (body (h1 "Hello, world!")
                     (p "The current date and time is "
                        ,(format-date "~H:~M:~S ~p ~m/~d/~Y"
                                      (date)))))))
    ))
```

This SURflet can be found in `howto/hello-date.scm`. The beginning of this SURflet is the same as in the previous example. The difference lies in the argument to `send-html/finish`. Note that the argument starts with a backquote (') rather than with a regular quote (') as in the previous example.

Instead of passing a "static" list, *i.e.* a list whose contents are given before execution, this SURflet uses the `quasiquote` and `unquote` feature of Scheme to create a "dynamic" list, *i.e.* a list whose contents are given only during execution. A "dynamic" list is introduced by a backquote (') and its dynamic contents are noted by `commata` (,). Thus, if the SURflet is executed while I am writing this `howto`, the argument to `send-html/finish` above is translated to

```
'(html (body (h1 "Hello, world!")
              (p "The current date and time is "
                 "13:09:03 PM 11/18/2003")))))
```

before it is passed to `send-html/finish`. Thus, using dynamic content can be easily done with Scheme's `quasiquote` and `unquote` feature. Of course, you can build your list in any way you want; the `quasiquote` notation is just a convenient way to do it.

### Several web pages in a row

The previous example SURflets only showed one page and finished afterwards. Here, we want to present two web pages in a row. We use the previously mentioned function `send-html/suspend`, which suspends after it has sent the page and continues when the user clicked for the next page. In contrast to `send-html/finish`, that expected SXML, `send-html/suspend` expects a function that takes an argument and returns SXML. The parameter the function gets (here: `k-url`) is the URL that points to the next page:<sup>2</sup>

```
(define-structure surflet surflet-interface
  (open surflets
        scheme-with-scsh)
  (begin

    (define (main req)
      (send-html/suspend
       (lambda (k-url)
         '(html (body (h1 "Hello, world!")
                      (p (a (@ (href ,k-url)) "Next page -->"))))))
      (send-html/finish
       '(html (body (h1 "Hello, again!")))))
    ))
```

This SURflet can be found in `howto/hello-twice.scm`. This example first displays a web page with the message “Hello, world!” and a link to the next page labeled with “Next page →”. When the user clicks on the provided link, `send-html/suspend` returns and the next statement after the call to `send-html/suspend` is executed. Here it is `send-html/finish` which shows a web page with the message “Hello, again!”.

When `send-html/suspend` returns, (almost) the complete context of the running SURflet is restored. Thus, every variable in the SURflet will retain its value during suspension. The consequence is that you don't have to worry about sessions, session variables and alike. The user can freely use the back

---

<sup>2</sup>In the API this URL is called the *continuation URL*.

button of her browser or clone a window while the SURflet will keep on responding in the expected way. This is all automatically managed by the SURflet-handler.

The only exception are variables whose values are changed by side effects, e.g. if you change a variable via `set!`. These variables keep their modified values, allowing communication between sessions of the same SURflet.<sup>3</sup>

### Begin and end of sessions

So far I don't have mentioned too much details about sessions. The reason is, as mentioned before, that the SURflet handler takes care of the session automatically as described in the previous paragraph.

The only thing you have to worry about is when your session *ends*. As long as your session hasn't been finished by `send-html/finish`, the user can move freely between the web pages your SURflet provides. Once you've finished the session via `send-html/finish`, this freedom ends. As the session is over, the user will get an error message when he tries to recall some web page from the server. The server will tell the user about the possible reasons for the error (namely that most likely the session was finished) and provides a link to the beginning of a new session.

Thus, `send-html/suspend` suspends the current execution of a SURflet, returning with the request for the next web page of your SURflet and `send/finish` finishes the session. The third sending function is `send-html` which just sends a web page. `send-html` does not return and does not touch the session of your SURflet instance.

### Abbreviations in SXML

The example in subsection "Several web pages in a row" wrote down the link to the next web page explicitly via the "`a`"-tag. As websites contain a lot of links, the sending functions (like `send-html/finish`) allow an abbreviation. The following SXML snippets are equivalent:

```
(a (@ (href ,k-url)) "Next page -->")
(url ,k-url "Next page -->")
```

`url` expects the target address as the next element and includes every text afterwards as part of the link.

---

<sup>3</sup>If you want to change a variable via side effects but you don't want to interfere with other sessions, you can use `set-session-data!` and `get-session-data`. See the API documentation in section 6.2 for further information.

There are also some other abbreviations. `(nbsp)` inserts `'&nbsp;'` into the HTML, `(*COMMENT* ...)` inserts a comment, and with `(plain-html ...)` you can insert arbitrary HTML code (*i.e.* strings) directly, without any string conversions. The last abbreviation, `surflet-form`, is discussed in the next section.

#### 6.1.4 How to write web forms

The SURflets come along with a library for easy user interaction. The following subsections will show how to write web forms and how to get the data the user has entered.

##### Simple web forms

Let's write a SURflet that reads user input and prints it out on the next page:

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scsh)
  (begin
    (define (main req)
      (let* ((text-input (make-text-field))
             (submit-button (make-submit-button))
             (req (send-html/suspend
                    (lambda (k-url)
                      '(html
                        (body
                          (h1 "Echo")
                          (surflet-form ,k-url
                                       (p "Please enter something:"
                                          ,text-input
                                          ,submit-button)))))))
             (bindings (get-bindings req))
             (user-input (input-field-value text-input bindings)))
        (send-html/finish
          '(html (body
                  (h1 "Echo result")
                  (p "You've entered: '" ,user-input "'")))))
      ))
```

Here are the details to the code in main:

```
(define (main req)
  (let* ((text-input (make-text-field))
         (submit-button (make-submit-button))
```

`make-text-field` and `make-submit-button` define two user interaction elements: a text input field and a submit button. SURflets represent user interaction elements by `Input-field` objects. Thus, user interaction elements are first class values in SURflet, unlike in many other web scripting languages, *e.g.* Java surflets, PHP or Microsoft Active Server Pages, *i.e.* you have a representation of a user interaction element in your program that you can pass to functions, receive them as return values, etc. You'll soon see the advantages of this approach.

```
(req (send-html/suspend
      (lambda (k-url)
        ' (html
            (body
              (h1 "Echo")
              (surflet-form ,k-url
                            (p "Please enter something:"
                                ,text-input
                                ,submit-button)))))))
```

Instead of discarding the return value of `send-html/suspend` as in the examples of the previous section, this time we'll save the return value, as it will contain the data the user has entered in our text input field.

The definition of the website is as described in the previous section except for the new abbreviation `surflet-form`. `surflet-form` creates the HTML code for a web form and expects as its next value the URL to the next webpage as provided by `send-html/suspend`, here named `k-url`. The remaining arguments constitute the content of the web form. Thus, the code above is equal to the following SXML:

```
(form (@ (action ,k-url) (method "GET"))
      (p "Please enter something:"
          ,text-input
          ,submit-button))
```

If you want to use the POST method instead of the default GET method, add the symbol 'POST after the URL:

```
(surflet-form ,k-url
              POST
              (p "Please enter something:"
                  ,text-input
                  ,submit-button))
```



The web page `send-html/suspend` sends to the browser looks like in figure [missing]. After the user has entered his data into the web form, `send-html/suspend` returns with the request object of the browser for the next page. This request object contains the data the user has entered.

```
(bindings (get-bindings req))
```

With the function `get-bindings` we pull out the user data of the request object. Here we save the user data into the variable `bindings`. `get-bindings` works for both request methods GET and POST.

```
(user-input (input-field-value text-input bindings)))
```

With the function `input-field-value` and the extracted user data we can read the value for an `input-field`. Here, we want to know what the user has entered into the `text-input-field`.

```
(send-html/finish
  '(html (body
    (h1 "Echo result")
    (p "You've entered: '" ,user-input "'."))))))
```

After we have extracted what the user has entered into the text field, we can show the final page of our SURflet and echo her input.

Thus, the scheme for user interaction is about the following:

- Create the user interaction elements, `input-fields`, you want to use in your web page.
- Send the web page with `send-html/suspend` to the browser. Plug in the `input-fields` in the web page as if they were usual values. Save the return value of `send-html/suspend`.
- Extract the user data from the return value of `send-html/suspend`.
- Read the values of each `input-field` out of the extracted user data with `input-field-value`.

The complete list of functions that create `input-fields` can be found in the API in section 6.2.

## Return types other than strings

As the user interaction elements are first class values in a SURflet, they can return other types than strings. For example the SURflets come with a number input field, *i.e.* an input field that accepts only text that can be interpreted as a number. If the user enters something that is not a number, `input-field-value` will return `#fas` the value of the number input field. If you'd rather want an error to be raised, you can use `raw-input-field-value` instead.

## Annotated input fields

The return value of an input field need not even be a primitive value. The SURflets library allows you to "annotate" your input fields with values which should be returned indicated by the user's input. *E.g.*, consider this SURflet:

```
(define-structure surflet surflet-interface
  (open surflets
    handle-fatal-error
      scheme-with-scsh)
  (begin
    (define (main req)
      (let* ((select-input-field
              (make-select
               (map make-annotated-select-option
                    '("Icecream" "Chocolate" "Candy")
                    '(1.5 2.0 0.5))))
             (req (send-html/suspend
                   (lambda (k-url)
                     '(html
                      (head (title "Sweet Store"))
                      (body
                       (h1 "Your choice")
                       (surflet-form
                        ,k-url
                        (p "Select the sweet you want:"
                          ,select-input-field
                          ,(make-submit-button)))))))
              (bindings (get-bindings req))
              (price (input-field-value select-input-field
                                         bindings)))
            (send-html/finish
             '(html (head (title "Receipt"))
                   (body
                    (h2 "Your receipt:")
                    (p "This costs you $" ,price "."))))))
      ))
```

Let's go through the important part of this SURflet:

```
(let* ((select-input-field
      (make-select
       (map make-annotated-select-option
            '("Icecream" "Chocolate" "Candy")
            '(1.5 2.0 0.5)))))
```

Here we define a select input field (a dropdown list). Instead of only providing a list of values that shall show up in the dropdown list and later examining which one was selected and looking up the price for the sweet, we bind the values in the list with the price while we create the select input field. When the select input field is shown in the browser, it will show the names of the sweets. When we lookup the user's input, we will get the associated price for the sweet. Again, this works not only with numbers, but with any arbitrary Scheme value (*e.g.* functions or records).

### Sending error messages

If a user tries to forge a SURflet-URL (*e.g.* by extracting the continuation URL from the HTML source and editing it), your SURflet has to deal with unexpected values. Usually, a forged SURflet-URL will result in an error that is raised in one of the SURflet library functions. If you don't catch this error, the SURflet handler will catch it for you, send an error message to the user *and terminate the current session* as your SURflet obviously encountered an unexpected error and might be in an invalid state. If you don't want this behavior, you can catch this error (like any other error that is raised by scsh) and send your own error message with `send-error` which is located in the `surfleets/error` package. The `handle-fatal-error` package can be useful in this context. Here's an example, that modifies the example from the previous subsection (modifications emphasized):

```
(define-structure surflet surflet-interface
  (open surflets
    handle-fatal-error
    surfleets/error
    scheme-with-scsh)
  (begin
    (define (main req)
      (let* ((select-input-field
            (make-select
             (map make-annotated-select-option
                  '("Icecream" "Chocolate" "Candy")
                  '(1.5 2.0 0.5)))))
```

```

(req (send-html/suspend
      (lambda (k-url)
        '(html
          (head (title "Sweet Store"))
          (body
            (h1 "Your choice")
            (surflet-form
              ,k-url
              (p "Select the sweet you want:"
                ,select-input-field
                ,(make-submit-button))))))
      (bindings (get-bindings req))
      (cost (with-fatal-error-handler
              (lambda (condition decline)
                (send-error (status-code bad-request)
                           req
                           "No such option or internal
                           error. Please try again.")(
              (raw-input-field-value select-input-field
                                     bindings))))))
(send-html/finish
 '(html (head (title "Receipt"))
        (body
          (h2 "Your receipt:")
          (p "This costs you $" ,cost ".")))))
))

```

Let's examine the important part of this example:

```

(cost (with-fatal-error-handler
      (lambda (condition decline)
        (send-error (status-code bad-request)
                     req
                     "No such option or internal
                     error. Please try again.")(
      (raw-input-field-value select-input-field
                           bindings))))

```

As mentioned in 6.1.4, this SURflet uses `raw-input-field-value` instead of `input-field-value` because the former raises an error while the latter returns `#f` in case of an error.

If a user forges a continuation URL, `raw-input-field-value` might not be able to find a valid value for the `select-input-field` and raises an error. This error is caught by the error handler which was installed by `with-fatal-error-handler`. The error handler uses `send-error` to send an error message to the browser. Its first argument is the status code of the error

message. See the documentation of the SUnet webserver for different status codes. The second argument is the request which was processed while the error occurred. The last argument is a free text message to explain the cause of the error to the user.

While in the original SURflet the user will still see the resulting receipt web page with an empty dollar amount and has her session finished, this modified version will show an error message and won't finish the session.

It is your choice, which version you choose, *i.e.* if you let the SURflet handler handle the occurring error automatically or if you install your own error handlers and use `raw-input-field-value`. However, be careful if you use `raw-input-field-value` along with check boxes. The HTML standard dictates that an unchecked check box does not appear in the data the browser sends to the server. Thus, `raw-input-field-value` won't find the check box in the data and raise an error which is not a "real" error as you might expect it.

### Your own input fields

The SURflet library contains constructors for all input fields that are described in the HTML 2.0 standard. See the SURflet API in section 6.2 for a complete list. The SURflet library also allows you to create your own input fields, *e.g.* an input field that only accepts valid dates as its input. This subsection gives you a short overview how to do this. You will find the details in the SURflet API.

Let's have a look at an SURflet that uses its own input field. The "input field", called nibble input field, consists of four check boxes which represent bits of a nibble (half a byte). The value of the input field is the number that the check boxes represent. *E.g.*, if the user checks the last two checkboxes, the value of the nibble input field is 3.

```
(define-structure surflet surflet-interface
  (open surflets
    surflets/my-input-fields
    scheme-with-scsh)
  (begin

    (define (make-nibble-input-fields)
      (let ((checkboxes (list (make-annotated-checkbox 8)
                           (make-annotated-checkbox 4)
                           (make-annotated-checkbox 2)
                           (make-annotated-checkbox 1)))))
        (make-multi-input-field
          #f "nibble-input"
          (lambda (input-field bindings)
            (let loop ((sum 0)
                       (checkboxes checkboxes))
              (loop (+ sum (checkbox-value checkbox))
                     (rest checkboxes)))))))
```

```

        (if (null? checkboxes)
            sum
            (loop (+ sum (or (input-field-value (car checkboxes)
                                                bindings)
                                0))
                  (cdr checkboxes))))))
'()
(lambda (ignore)
  checkboxes))))

(define nibble-input-field (make-nibble-input-fields))

(define (main req)
  (let* ((req (send-html/suspend
              (lambda (new-url)
                '(html (title "Nibble Input Widget")
                      (body
                        (h1 "Nibble Input Widget")
                        (p "Enter your nibble (msb left):")
                        (surflet-form ,new-url
                                    ,nibble-input-field
                                    ,(make-submit-button)))))))
        (bindings (get-bindings req))
        (number (input-field-value nibble-input-field bindings)))
    (send-html
     '(html (title "Result")
           (body
            (h2 "Result")
            (p "You've entered " ,number "."))))))))
))

```

Let's go through this SURflet step by step.

```

(define-structure surflet surflet-interface
  (open surflets
    surflets/my-input-fields
    scheme-with-scsch)

```

If you want to create your own input fields, you have to open the surflets/my-input-fields package.

```

(begin
  (define (make-nibble-input-fields)
    (let ((checkboxes (list (make-annotated-checkbox 8)
                          (make-annotated-checkbox 4)
                          (make-annotated-checkbox 2)
                          (make-annotated-checkbox 1)))))

```

`make-nibble-input-fields` is the constructor for our new type of input field. As mentioned before, we use check boxes to let the user enter the nibble. We use annotated checkboxes for this purpose whose value is the value in the nibble.

```
(make-multi-input-field
  #f "nibble-input")
```

The value of our new input field will depend on the value of several real input fields. Thus we create a multi input field. If the value depended only on the browser data that is associated to the name of our input field, we would use `make-input-field` instead, which creates a usual input field. *E.g.*, if we wanted to create a date input field that accepts only valid dates as input and used a text input field for this purpose, we would use `make-input-field`.

The first two parameters is the name of the input field and its type. As we use checkboxes to represent our input field, we don't need the name field. The type field is meant for debugging purposes, so you can identify the type of the input field during a debugging session.

```
(lambda (input-field bindings)
  (let loop ((sum 0)
             (checkboxes checkboxes))
    (if (null? checkboxes)
        sum
        (loop (+ sum (or (input-field-value (car checkboxes)
                                             bindings)
                          0))
              (cdr checkboxes)))))
```

The next parameter is the so called transformer function. `raw-input-field-value` calls the transformer function to determine the value of the input field depending on the given bindings. The transformer function of a multi input field (which our nibble input field is) gets the input field and the bindings as parameters. A usual input field would only get the data that is associated to its name.

The transformer function of our nibble input field goes over each check box, looks it up in the bindings and adds its value to a sum, if `input-field-value` can find it. If it can't find it, zero is added instead. The value of our nibble input field is the resulting sum.

The rest of the SURflet is straight forward and not repeated here again. We create, use and evaluate the nibble input field as we do with every other input field.

### 6.1.5 Program flow control

With the techniques shown so far it is rather difficult to create a web page that has several different successor webpages rather than only one web page. This section will show you how to do this with the SURflets. Basically, there are two different methods how to perform this task. One method is to mark each link in some way and evaluate the mark after `send-html/suspend` has returned. The other method is to bind a callback function to each link that is called when the user selects the link. This section shows both methods.

#### Dispatching to more than one successor web page

The basic idea of dispatching is to add a mark to a link and evaluate it after the user has clicked on a link and `send-html/suspend` returned. Let's have a look at an example. It shows an entry page at which the user states the language in which she wants to be greeted:

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scsch)
  (begin

    (define (main req)
      (let* ((english (make-address))
             (german (make-address))
             (req (send-html/suspend
                   (lambda (k-url)
                     '(html
                       (head (title "Multi-lingual"))
                       (body
                        (h2 "Select your language:")
                        (ul
                         (li (url ,(english k-url) "English")
                           (li (url ,(german k-url) "Deutsch"))))))))
                   (bindings (get-bindings req)))
             (case-returned-via bindings
              ((english) (result-page "Hello, how are you?"))
              ((german) (result-page "Hallo, wie geht es Ihnen?")))))

      (define (result-page text)
        (send-html/finish
          '(html
            (head (title "Greeting"))
            (body
             (h2 ,text)))))

    ))
```



Let's see how main presents the different options:

```
(define (main req)
  (let* ((english (make-address))
        (german (make-address)))
```

Of course you don't have to worry about adding the mark to the links. Instead, we create the links with make-address.

```
(req (send-html/suspend
      (lambda (k-url)
        ' (html
            (head (title "Multi-lingual"))
            (body
              (h2 "Select your language:")
              (ul
                (li (url ,(english k-url) "English")
                  (li (url ,(german k-url) "Deutsch"))))))))))
```

make-address returns a function you can call to create the link as we did here with

```
(li (url ,(english k-url) "English")
```

This creates a list item which contains a hyperlink labeled "English". The hyperlink is created by the SXML abbreviation url as shown in 6.1.3. Instead of just passing the continuation URL k-url to url, we create the marked link by calling the function make-address gave us.

```
(bindings (get-bindings req)))
(case-returned-via bindings
  ((english) (result-page "Hello, how are you?"))
  ((german) (result-page "Hallo, wie geht es Ihnen?"))))
```

After send-html/suspend has returned, we can evaluate which link the user has clicked by using case-returned-via. case-returned-via works similar to the regular case of Scheme. It evaluates the body of the form whose initial list contains the address that the user used to leave the website. *E.g.*, if the user has selected "German" as her preferred language and clicked on the link we have named german in our SURflet, case-returned-via will evaluate its second form and the SURflet will display the greeting in German.

case-returned-via is syntactic sugar like the regular case. However, instead of equal? it uses returned-via. returned-via takes the bindings

and an address and returns #t, if the user left the web page via this address (*i.e.*, via the link that is represented by this address) and #f otherwise. `returned-via` does not end with a question mark as it might return other values as well as we will see shortly. Of course, it is your choice if you want to use `case-returned-via` or explicitly `returned-via`.

## Annotated dispatching

The approach shown in the previous subsection has one major drawback: the meaning of an address becomes clear only when you look at the dispatching section of `case-returned-via`. This subsection shows you how to link the meaning and the representation of an address closer together.

We modify the previous code example slightly to this SURflet (differences emphasized):

```
(define-structure surflet surflet-interface
  (open surflets
    scheme-with-scsh)
  (begin

    (define (main req)
      (let* ((language (make-annotated-address))
             (req (send-html/suspend
                    (lambda (k-url)
                      ' (html
                        (head (title "Multi-lingual"))
                        (body
                          (h2 "Select your language:")
                          (ul
                            (li (url ,(language k-url
                                                "Hello, how are you?")
                                "English")
                                (li (url ,(language k-url
                                                "Hallo, wie geht es Ihnen?")
                                "Deutsch"))))))))
              (bindings (get-bindings req)))
            (case-returned-via bindings
              ((language) => result-page))))

    (define (result-page text)
      (send-html/finish
        ' (html
          (head (title "Greeting"))
          (body
            (h2 ,text))))))
  ))
```

Let's look at the differing parts:

```
(let* ((language (make-annotated-address))
```

To link the meaning with the address itself, we use an annotated address. As we can annotate the address now, we don't need two distinct addresses anymore.

```
(li (url ,(language k-url
                    "Hello, how are you?")
    "English")
  (li (url ,(language k-url
                    "Hallo, wie geht es Ihnen?")
    "Deutsch"))))))))
```

In addition to the continuation URL `k-url` we also annotate the address with a value. Here we use the different greetings as the annotation. The address can be annotated with any arbitrary Scheme value, *e.g.* functions or records.

```
(case-returned-via bindings
  ((language) => result-page)))
```

`case-returned-via` has an extended syntax similar to `cond` that it useful with annotated address. The arrow `'=>'` calls the following function with the annotation of the address via which the user has left the web page. You can extract the annotation yourself with `returned-via` like this:

```
(result-page (returned-via language bindings))
```

This will call `result-page` with the annotation of the address via which the user has left the web page. `returned-via` returns `#f`, if the user didn't leave the web page via one of the links created with this address (which is not really possible in this example).

## Callbacks

The other method to lead to different successor web pages is using callbacks. A callback is a function that is called if the user leaves the web page via an associated link. This is different from the dispatch method where `send-html/suspend` returns. You can create a web page that only uses callbacks to lead to successor web pages and you don't have to use `send-html/suspend`. Instead, you can use `send-html`.

Although it is possible to use several different callbacks in a single web page, this is not recommended. The reason lies in the implementation of a callback, which saves the current continuation. Several different callbacks would result in the storage of the several slightly different continuations. This is unnecessary, as you can annotate the callbacks with the arguments for the callback function. Let's see an example which is a variation of the previous examples (important parts / differences emphasized):

```
(define-structure surflet surflet-interface
  (open surflets
    surflets/callbacks
    scheme-with-scsh)
  (begin

    (define (main req)
      (let ((language (make-annotated-callback result-page)))
        (send-html
          '(html
            (head (title "Multi-lingual"))
            (body
              (h2 "Select your language:")
              (ul
                (li (url ,(language "Hello, how are you?")
                      "English")
                  (li (url ,(language "Hallo, wie geht es Ihnen?")
                          "Deutsch")))))))))

    (define (result-page req text)
      (send-html/finish
        '(html
          (head (title "Greeting"))
          (body
            (h2 ,text)))))
  ))
```

The differences to the dispatch method are the following: you have to open the `surflets/callbacks` package to use callbacks, you don't use the continuation URL to create the callback link, and the callbacked function must accept the request from the browser as the first argument. Furthermore, you don't have to use `send-html/suspend`, if a user can only leave your web page via callbacks. However, it can be sensible to combine the dispatch and the callback method, in which case you have to use `send-html/suspend`.

Note that is nonsensical to create a callback on top level, *i.e.* the call to `make-annotated-callback` must occur every time `main` is called and not only once when the `SURlet` is read into memory. For the same reason it is nonsensical in most cases to reuse a callback.

The SURflet library provides also a wrapper function with which you can instruct the callback to call different functions instead of a single one. If you create your callback like

```
(let ((callback (make-annotated-callback callback-function)))  
  ...)
```

you can instruct the callback to call different functions like this:

```
(callback function1 arg1 arg2)  
...  
  
(callback function2 arg3 arg4 arg5)
```

Again, it is your choice which option you want to use. Note that calling a function with several arguments and of different amount each time is also possible if you only use a single function for the callback.

### 6.1.6 Data management

When you write web programs, there are usually two kinds of data that you use: data that is local to each instance of a SURflet, *e.g.* the user's login, and data that is global to each instance of a SURflet, *e.g.* a port to a logfile. Changes to local data is only visible to each session of a SURflet, while changes to global data is visible to every session of a SURflet.

The SURflet library does not really distinguish between these two types of data, but provides ways to realize both of them in a convenient way that is not (really) different from the way you handle these data types in a regular Scheme program.

If a data item is globally used in your SURflet, define it global (on top level) and change its values with `set!`. If a data item is locally used, define it locally (in your function) and do not change its value with `set!`.

If the following sounds too technical to you, you can safely skip this paragraph. The reason why the distinction between global and local data is done via whether you mutate the data's value with `set!` is that the SURflets are implemented with continuations. Continuations cannot reflect changes that are done via `set!` (or side effects in general) and thus such changes are globally visible. On the other hand continuations represent states of a program and a reified continuations reifies also the values of all data.

But what to do if you happen to want to change your *local* data's value with `set!`? The SURflet library provides a place where you store such mutable local data and two functions to access it: `set-session-data!` sets the mutable local

data and `get-session-data` reads the mutable local data. Here is an example. It uses the callback technique that was presented in the previous section. If you haven't read that section, you only need to know that `show-page` is called again and again as long as the user keeps on clicking on "Click".

```
(define-structure surflet surflet-interface
  (open surflets
    surflets/callbacks
    scheme-with-scssh)
  (begin
    (define (main req)
      (set-session-data! -1)
      (let ((start (make-annotated-callback show-page)))
        (show-page req 'click start)))

    (define (show-page req what callback)
      (if (eq? what 'click)
          (click callback)
          (cancel)))

    (define (click callback)
      (set-session-data! (+ 1 (get-session-data)))
      (send-html
        '(html
          (head (title "Click counter"))
          (body
            (h2 "Click or cancel")
            (p "You've already clicked "
              ,(get-session-data)
              " times.")
            (p (url ,(callback 'click callback) "Click")
              (url ,(callback 'cancel callback) "Cancel"))))))

    (define (cancel)
      (send-html/finish
        '(html
          (head (title "Click counter finished"))
          (body
            (h2 "Finished")
            (p "after " ,(get-session-data) " clicks."))))))
  ))
```

At the beginning of `main` we initialize the mutable local data with `set-session-data!`.

```
(define (main req)
  (set-session-data! -1)
```

```
(let ((start (make-annotated-callback show-page)))
  (show-page req 'click start)))
```

Afterwards, we create and save a callback that will be called again and again. We call `show-page` with the callback to show the first web page.

```
(define (show-page req what callback)
  (if (eq? what 'click)
      (click callback)
      (cancel)))
```

`show-page` evaluates its second argument `what` to determine what to do next: continue or cancel.

```
(define (click callback)
  (set-session-data! (+ 1 (get-session-data)))
  (send-html
   '(html
     (head (title "Click counter"))
     (body
      (h2 "Click or cancel")
      (p "You've already clicked "
         ,(get-session-data)
         " times.")
      (p (url ,(callback 'click callback) "Click")
          (url ,(callback 'cancel callback) "Cancel"))))))
```

If the user continues, `click` increases the mutable local counter and shows the next page.

Note that we don't use `send-html/suspend` here because we use the callback to lead to the next web page. If the user clicks on the link labeled with "Click" or "Cancel", `show-page` will be called with `'click` or `'cancel`, respectively, and the callback as parameters. This creates an endless loop without saving endless states of the `SURflet`.

`cancel` shows the final page with the amount of clicks performed.

### 6.1.7 My own SXML

Section 6.1.3 introduced SXML, the way how `SURflets` represent HTML. This section will show you, how you can create your own rules to translate from SXML to HTML.

## Terms and theoretical background

This subsection will introduce the main concepts of the translation process and some necessary terms we will use in the following.

The translation process from SXML to HTML takes two steps. In the first step, SXML is translated to an intermediate form. This is done by the *translator*. In the second step, the intermediate form is printed into an HTML string. This is done by the *printer*. The intermediate form looks very much like SXML, but contains only *atoms* or, recursively, list of atoms. Atoms are numbers, characters, strings, `#f`, and the empty list. We call the intermediate form an *atom tree* and the list from which we've started an *SXML tree*.

The basic unit in the translation process is a *conversion rule*. A conversion rule consists of a trigger and a conversion function. As its first element, the trigger identifies the list for which the translator shall call the conversion function. The translator calls the conversion function with all list elements as parameters and replaces the whole list by the result of the conversion function. The result of the conversion function is supposed to be an atom tree.

The translator takes the SXML tree and a list of conversion rules as arguments. It then traverses the SXML tree depth first and calls the conversion functions according to the triggers it encounters, replacing the nodes in the SXML tree with the return values of each conversion function called. The result of this translation step will be an atom tree, which the printer will print into a string or port.

The translator calls the conversion function in two different modes, depending on the conversion rule. The regular mode is the *preprocess* mode: the translator translates every argument of the conversion function before calling it. The other mode is the *unprocessed* mode: the translator calls the conversion function directly without preprocessing the arguments. This is, the translator stops traversing the SXML tree at nodes that trigger a conversion rule in unprocessed mode.

There are two default triggers which you can't use in your translation rules: `*default*` and `*text*`. `*default*` as the trigger marks the default conversion rule which the translator uses if no other conversion rule triggers. `*text*` marks the text conversion rule and triggers, if the node in the SXML tree is a string. In the standard conversion rule set the text conversion rule performs HTML escaping, e.g. for the ampersand (&).

## Outlook

More to come soon about SURflets consisting of different parts and individual SXML.



## 6.2 API description

The SURflet server comes with an extensive set of modules. This section describes the modules and the programming interfaces. See the `howto` section 6.1 for a practical guide. Note that most of the procedures mentioned here are meant to be called from within a SURflet.

### 6.2.1 The SURflet server

The SURflet server provides basic procedures to send web content to a client. To enable the SUNet webserver to serve SURflets, you have to add the SURflet handler to it, which resides in the `surflet-handler` structure. *E.g.:*

```
(httpd
  (make-httpd-options
    ...

    with-request-handler
    (alist-path-dispatcher
      (list
        (cons "surflet" (surflet-handler (with-surflet-path
                                          "web-server/root/surflets"))))
        (rooted-file-or-directory-handler "web-server/root/htdocs"))))
```

This will set up the SURflet handler to handle requests directed to the directory `/surflet/`. The SURflet handler can only handle requests directed to SURflets. Here's the interface description:

`(surflet-handler options)`  $\longrightarrow$  *request-handler* procedure

This procedure sets up the SURflet handler and returns the according request handler for the SUNet webserver. The `options` argument is similar to the one passed to `httpd` and is explained below.

The SURflet handler accepts requests (solely) to SURflets whose file name must have the extension `.scm`. The structure of SURflets is explained below. The SURflet handler receives the `request` from `httpd`, translates it to a `surflet-request` and passes it to the requested SURflet. The SURflet in turn is expected to return a `surflet-response`, which the SURflet handler translates to a `response` for `httpd`. Thus, the SURflet deals only with `surflet-request` and `surflet-response` objects. The structure of these objects and how they are passed around is explained below.

A SURflet may also return a `redirect` response, which the SURflet handler passes to the `httpd` untouched. See 2.3 for details.

The SURflet handler calls the SURflet wrapped into an error handler that catches any error the SURflet may yield. In this case, it terminates the SURflets session (see below for more on sessions) and returns an error response to httpd with the error code 500 “Internal Server Error” and a description of the error.

The *options* argument can be constructed in a similar way to the options argument of httpd. The procedures’ names are of the form *with-...* and they all either create a new option or add a new parameter to a given option. The new parameter is always the first argument while the (old) option the optional second one. The following procedures reside in the *surflet-handler/options* structure.

(*with-surflet-path surflet-path [options]*)  $\longrightarrow$  *options* procedure

This specifies the path in which the SURflet handler looks for SURflets. The *surflet-path* is a string. This option must be given for the handler to work.

(*with-session-lifetime seconds [options]*)  $\longrightarrow$  *options* procedure

This specifies the initial lifetime of a session. The lifetime of a session is the number of seconds the SURflet handler waits for a response from a client for that session, before she automatically finishes it. See below for details on sessions. Defaults to 600, *i.e.* 10 minutes.

(*with-cache-surflets? cache-surflets? [options]*)  $\longrightarrow$  *options* procedure

This specifies whether the SURflet handler caches SURflets. The caching of SURflets is a prerequisite for SURflet wide global variables. See below for details on the scope of variables in SURflets. Defaults to #t.

(*with-make-session-timeout-text timeout-text-procedure [options]*)  $\longrightarrow$  *options* procedure

This specifies a procedure that generates the timeout text. The SURflet handler displays the timeout text when she receives a request for a SURflet session that does not exist, either because the SURflet finished its session, the session has timed out, or the URL is illformed. The default is an English text with an explanation of the possible reasons and a link to a new session of the requested SURflet. *timeout-text-procedure* accepts the string path to the SURflet that was requested and returns a string.

Similar to the httpd options there exists a procedure to avoid parenthesis:

(*make-surflet-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-surflet-options* returns the resulting options value.

For example,

```
(surflet-handler
  (make-surflet-options
    with-surflet-path "root/surflets"
    with-session-timeout 3600))
```

defines the SURflet handler to serve SURflets from the directory `root/surflets` and to timeout unused sessions after one hour.

The SURflet handler allows runtime read and write access to her options:

```
(options-surflet-path options) → string           procedure
(options-session-lifetime options) → integer       procedure
(options-cache-surflets? options) → boolean         procedure
(options-make-session-timeout-text options) → procedure procedure
```

These procedures return the stored value for the respective option. See above for the description of the options.

```
(set-options-surflet-path! options surflet-path) → undefined procedure
(set-options-session-lifetime! options seconds) → undefined procedure
(set-options-cache-surflets?! options cache-surflets?) → undefined procedure
(set-options-make-session-timeout-text! options timeout-text-procedure) → undefined procedure
```

These procedures change the respective option value. See above for the description of the arguments. Note that changing the *surflet-path* within a SURflet may result in the SURflets being unreachable. Turning the cache off will not empty the SURflet cache.

## 6.2.2 SURflets

Technically, SURflets are Scheme48 structures, that have the name `surflet` and export a main procedure. The file in which their definition reside must have the extension `.scm`. The main procedure must accept the initial `surflet-request` as an argument. She may or may not return, but if she does, she must return either a `surflet response` or a `redirect response`. For example, this is a valid SURflet definition:

```
(define-structure surflet surflet-interface
  (open scheme-with-scsch
    surflets)
  (begin
    (define (main req)
      (send-html/finish
        '(html (body (p "Hello world!")))))
  ))
```

`surflet-interface` is a predefined interface description that exports the main procedure. It is recommended to use this interface. `surfleets` is a structure that combines the most commonly used structures to write SURflets. `send-html/finish` is one of the procedures that sends HTML to the client. More on this below.

As SURflets are Scheme48 structures, you can use all capabilities of the Scheme48 module language. See the documentation of Scheme48 for details.

SURflets should not use the `shift-reset` structure, as this might confuse the SURflet handler. The use of threads within a surflet is currently discouraged, as some procedures might not work, especially procedures dealing with session IDs.

### 6.2.3 SURflet management

Upon an initial client request, the SURflet handler looks for the requested SURflet, loads it dynamically, installs an error handler and calls the main function of the SURflet with the initial `surflet-request`. To minimize the time of loading a SURflet, the SURflet handler caches the structure of the SURflet in a cache, the SURflet cache. As the SURflet is cached, its global values will remain unchanged even through times when there are no active sessions of the SURflet. Changing global SURflet values is a possibility to exchange data between different sessions of the same SURflet. Note that you have to take care to serialize the access to commonly shared, mutated data.

The SURflet handler allows access to its cache via the following procedures. The access to these procedures is currently unrestricted but may be restricted in future versions of the SURflet server.

`(get-loaded-surfleets)`  $\longrightarrow$  *list* procedure

This returns a list of the file names of the loaded SURflets.

`(unload-surflet surflet)`  $\longrightarrow$  *undefined* procedure

This removes the *surflet* from the SURflet cache. The *surflet* is identified by its file name, as returned from `get-loaded-surfleets`.

`(reset-surflet-cache!)`  $\longrightarrow$  *undefined* procedure

This empties the SURflet cache.

Of course, when a SURflet is removed from the cache, the values in its sessions remain untouched. However, if the SURflet is newly loaded into the SURflet cache, the SURflet handler treats it like a new SURflet, *i.e.* the sessions of the "old" and the "new" SURflet (though physically the same) do *not* share their global data in any way.

### 6.2.4 Surflet Request

SURflets get their input from `surflet-request` objects. The relevant procedures are the following. They are all exported by the `surflet-handler/requests` alias `surflet-requests` structure.

<code>(surflet-request? object)</code>	$\longrightarrow$ <i>boolean</i>	procedure
<code>(surflet-request-method surflet-request)</code>	$\longrightarrow$ <i>string</i>	procedure
<code>(surflet-request-input-port surflet-request)</code>	$\longrightarrow$ <i>input-port/undefined</i>	procedure
<code>(surflet-request-uri surflet-request)</code>	$\longrightarrow$ <i>string</i>	procedure
<code>(surflet-request-url surflet-request)</code>	$\longrightarrow$ <i>url</i>	procedure
<code>(surflet-request-version surflet-request)</code>	$\longrightarrow$ <i>pair</i>	procedure
<code>(surflet-request-headers surflet-request)</code>	$\longrightarrow$ <i>alist</i>	procedure

The procedures inspect `surflet-request` values. Most of them return the values of the underlying request object from `httpd`. `surflet-request?` is a predicate for surflet requests. `surflet-request-method` extracts the method of the HTTP request; it's a string and either "GET" or "POST". SURflets won't receive requests with other methods. `surflet-request-input-port` returns an input-port that contains data from the client on POST requests and that the SURflet can safely read. If the request is no POST request, its value is undefined. `surflet-request-uri` returns the escaped URI string as read from the request line. `surflet-request-url` returns the respective HTTP URL value (see the description of the `url` structure in chapter 4). `surflet-request-version` returns a (major . minor) integer pair representing the version specified in the HTTP request. `surflet-request-headers` returns an association lists of header field names and their values, each represented by a list of strings, one for each line.

For some unknown weird cases, there are also these two procedures:

<code>(surflet-request-socket surflet-request)</code>	$\longrightarrow$ <i>socket</i>	procedure
<code>(surflet-request-request surflet-request)</code>	$\longrightarrow$ <i>request</i>	procedure

`surflet-request-socket` returns the socket connected to the client. As with requests, SURflets should not perform I/O on this socket. See section 2.2 for reasoning. `surflet-request-request` allows access to the underlying request object from `httpd`. Both procedures should not be necessary in normal operation and their usage is discouraged.

### 6.2.5 Surflet Response

SURflets answer to a request by sending a `surflet-response` to the SURflet handler. The next section deals with how the surflet responses are sent to

the SURflet handler. The relevant procedures for `surflet-response` are the following. They are all exported by the `surflet-handler/responses` alias `surflet-response` structure.

`(make-surflet-response status content-type headers data) → surflet-response` procedure

This creates a `surflet-response`. *status* is the status code of the response. See section 2.3 for details on this. *content-type* is the MIME type of the data, *e.g.* "text/html". *headers* is an association list of headers to be added to the response, each of which consists of a symbol representing the field name and a string representing the field value. *data* is the actual data. It must be either a string or a list of values that will be displayed.

`(valid-surflet-response-data? object) → boolean` procedure

This is a predicate on objects that may be surflet data, *i.e.* a string or a list (of objects that will be displayed).

`(surflet-response? object) → boolean` procedure

`(surflet-response-status surflet-response) → status-code` procedure

`(surflet-response-content-type surflet-response) → string` procedure

`(surflet-response-headers surflet-response) → alist` procedure

`(surflet-response-data surflet-response) → surflet-data` procedure

The procedures return `surflet-response` values. `surflet-response?` is a predicate for `surflet-responses`. `surflet-response-status` returns the status code of the response. See section 2.3 for details on the status code. `surflet-response-content-type` returns the MIME type of the response. `surflet-response-headers` returns the association list of header field names and their values, each represented by a list of strings, one for each line. `surflet-response-data` returns the data of the `surflet-response`, the actual answer of the SURflet.

## 6.2.6 Sessions

A session is a set of web pages that logically belong together, *e.g.* a user surfing through her webmail. A session starts with the initial request for a SURflet and ends either explicitly by the SURflet, or implicitly after a timeout. A third, not so common case is its deletion from the session table.

The procedures presented in this subsection are all accessible via the `surflets/sessions` structure.

### Session management

The SURflet handler automatically manages the sessions for each SURflet, thus the SURflet does not have to deal with sessions or state control (as it is the case

with most other programming interfaces for web applications). In particular, a SURflet does not have to take care of saving the contents of variables before emitting a web page and restoring the values later upon the next request, or determining how far the user has proceeded in the application. The only thing a SURflet may want to do is to tell the SURflet handler when a session finished by calling `send/finish` or an equivalent procedure (see below).

Although a SURflet does not have to deal with the management of the sessions, the SURflet handler allows access to its management structures.

`(instance-session-id)`  $\longrightarrow$  *session-id* procedure

This returns the session ID for the current session. The current session is the session from which the function is called. The SURflet handler guarantees that there won't be two sessions with the same ID for any given point in time. However, session IDs may be reused.

`(get-session session-id)`  $\longrightarrow$  *session* procedure

This returns the session for the given session ID.

`(get-sessions)`  $\longrightarrow$  *alist* procedure

This returns the complete list of all active session of the SURflet handler. The list is an association list with the session-id as key and the session as value.

The access to this procedure is currently unrestricted but may be restricted in future versions of the SURflet server.

`(delete-session! session)`  $\longrightarrow$  *undefined* procedure

This deletes the specified session from the session table. Future requests to the session are answered with the timeout text.

`(session-alive? session)`  $\longrightarrow$  *boolean* procedure

This returns `#t`, if the specified session is alive, *i.e.* requests to it will be answered by the appropriate SURflet. Otherwise, she returns `#f`.

`(session-surflet-name session)`  $\longrightarrow$  *string* procedure

`(session-session-id session)`  $\longrightarrow$  *session-id* procedure

These procedures inspect values of a session. `session-surflet-name` returns the name of the SURflet for which the session was created. `session-session-id` returns the session ID of the session.

For each session, the SURflet handler has a counter running. She resets the counter each time she receives a request for the session. When the counter reaches a particular number of seconds, the lifetime of the session, the SURflet

handler deletes the session and removes it from its session table. She will answer all future requests for the session with the timeout text. The following procedures deal with the lifetime of a session.

```
(session-lifetime session)  → integer      procedure
(set-session-lifetime! session new-lifetime) → undefined procedure
```

`session-lifetime` returns the number of seconds the SURflet handler will initially wait before she automatically finishes the session. `set-session-lifetime!` changes the initial lifetime of the *session* to *new-lifetime* and also resets the counter for that session.

```
(session-adjust-timeout! session [lifetime]) → undefined procedure
(adjust-timeout! [lifetime]) → undefined procedure
```

These reset the counter for the lifetime of either the given *session* (`session-adjust-timeout!`) or the current session (`adjust-timeout!`). Both procedures give the session a lifetime of either *lifetime* seconds or of the lifetime seconds stored for the according *session*.

In order to allow easy web programming, the SURflet handler automatically saves and reifies continuations of a session. This is totally transparent to the web programmer. For administration purposes, the SURflet handler offers access to the continuation table of a session via the following procedures.

```
(session-continuation-table session) → table      procedure
(session-continuation-table-lock session) → lock   procedure
(session-continuation-counter session) → thread-safe-counter procedure
```

These functions return the continuation table, the lock for the continuation table and the counter for the continuations, respectively. The continuation *table* is a hash table with the continuation ID as key and the continuation as value, based on the tables structure of Scheme48. The *lock* is based on the locks structure of Scheme48. The *thread-safe-counter* is based on the thread-safe-counter structure that is part of the SURflets.

The access to these functions is currently unrestricted but may be restricted in future versions of the SURflet server.

The surflets/continuations structure also offers procedures to access the continuations.

```
(get-continuations session) → list      procedure
```

Returns a list of all continuations of the *session*. The list elements are pairs with the car being the session and the cdr being the continuation.

```
(delete-continuation! session-continuation) → undefined procedure
```



Removes the specified continuation from the continuation table. *session-continuation* is a pair as returned from `get-continuations`. It is no error if the session or the continuation does not exist anymore.

The access to this functions is currently unrestricted but may be restricted in future versions of the SURflet server.

`(continuation-id session-continuation) → number` procedure

Returns the continuation ID of the continuation specified by *session-continuation* which is a pair as returned by `get-continuations`.

The `surfleets/ids` structure provides procedures to determine the session and continuation IDs of the current session. See also the entry for `resume-url-ids` somewhere else in this document.

`(my-session-id surlfet-request) → number` procedure

`(my-continuation-id surlfet-request) → number` procedure

`(my-ids surlfet-request) → number number` procedure

These return the session and continuation ID that where used to access the current session. The procedures work for every *surlfet-request* except for the initial one that main gets. The values returned by `my-ids` are the session and the continuation ID in this order.

`(surflet-file-name surlfet-request) → string` procedure

This returns the name of the SURflet of the current session.

## Session data

The SURflet handler distinguishes three kinds of **session data**: session data that is local to a session of a SURflet and not mutated, session data that is local to a session of a SURflet and mutated and session data that is global to all sessions of a SURflet. Every variable value that is never mutated is automatically local to the session of the SURflet. Variable values that are mutated are automatically global to all sessions of the SURflet. Values that have to be mutated but should be local to a session of the SURflet must be stored in a special place, the session data field of the SURflet handler.<sup>4</sup>

`(get-session-data) → object` procedure

`(set-session-data! new-value) → undefined` procedure

---

<sup>4</sup>The reason for this distinction is the fact that the SURflet handler saves and reifies the continuation of a SURflet to realize the easy programming of web applications. Mutations of values remain visible after reifying the continuation.

These procedures allow read/write access to the session data field of the SURflet handler. The SURflet handler installs a session data field for each session she creates. `get-session-data` reads the contents of this field, which is initially `#f`. `set-session-data!` sets the contents of the field to *new-value*. Mutations to the values, no matter when they occur, are local to the session from which the procedures are called, *i.e.* the changes are only visible within a particular session of the SURflet. The procedures are exported by the `surflet-handler/session-data` structure.

### 6.2.7 Basic I/O

The SURflet communicates with the web client basically with the following send primitives. They are exported by the `surflet-handler/primitives` structure along with the procedures from `surflet-handler/requests`, `surflet-handler/responses` and the `status-code` syntax.

`(send surflet-response)`  $\longrightarrow$  *no return value* procedure

This procedure sends the data of the *surflet-response* to the client and does not return.

`(send/finish surflet-response)`  $\longrightarrow$  *no return value* procedure

This procedure does the same as `send`, but it also finishes the session of the SURflet. Future requests to the SURflet will be answered with the timeout text (see above).

`(send/suspend surflet-response-maker)`  $\longrightarrow$  *surflet-request* procedure

This procedure suspends the current computation of the SURflet and calls *surflet-response-maker* with the continuation-URL of the current session to create the actual *surflet-response*. When a client requests the continuation-URL, the computation of the SURflet will resume with `send/suspend` returning that request of the client. See 6.2.9 for details on continuation-URLs.

`(send-error status-code surflet-request [messages])`  $\longrightarrow$  *no return value* procedure

This sends an error response to the client. *status-code* is the status code of the error, see section 2.3 for details. *surflet-request* is the last *surflet-request* the SURflet received; if unknown this argument may be `#f`. *messages* may contain some further information about the cause of the error.

## 6.2.8 Web I/O

Most of the time, a SURflet won't send arbitrary data but HTML to the client. For this purpose, the SURflets provide extensive support. First of all, they provide procedures specially designed for submitting HTML.

<code>(send-html <i>sxml</i>)</code>	$\longrightarrow$ <i>no return value</i>	procedure
<code>(send-html/finish <i>sxml</i>)</code>	$\longrightarrow$ <i>no return value</i>	procedure
<code>(send-html/suspend <i>sxml-maker</i>)</code>	$\longrightarrow$ <i>surflet-request</i>	procedure

These are the equivalent procedures to the `send` primitives. `send-html` and `send-html/finish` accept an SXML object (more on that below), translate it into HTML and send it to the client, the latter finishing the session. `send-html/suspend` suspends the current computation, calls *sxml-maker* with the continuation-URL, translates the resulting SXML into HTML and sends it to the client. When the client requests the continuation-URL, the computation is resumed and `send-html/suspend` returns with the *surflet-request*.

### SXML

For easy creation of HTML output, the `send-html` procedures mentioned above represent the HTML in SXML. SXML is a creation of Oleg Kiselyov. Basically, SXML is a list whose *car* is an SXML tag, a symbol representing the HTML tag, and the *cdr* are other SXML elements that will be enclosed by the HTML tag. For example,

```
'(h2 "Result")
```

represents the tag `h2`, that encloses the text "Result". The represented HTML is

```
<h2>Result</h2>.
```

As in HTML, elements may be nested:

```
'(body (h2 "Result") (p "Example"))
```

represents

```
<body><h2>Result</h2><p>Example</p>.
```

The `@` symbol marks HTML attributes. The attributes follow the `@` symbol in two element lists, the first element being the name of the attribute and the last its value. For example, this is a link:

```
'(a (@ (href "add-surflet.scm") (name "linklist"))) "Make new  
calculation."
```

representing the following HTML

```
<a href="add-surflet.scm" name="linklist">Make new
calculation.</a>.
```

The attributes form will only be recognized as such if it is the second element of a list, right after the SXML tag.

```
(sxml-attribute? object)  → boolean           procedure
(sxml-attribute-attributes sxml-attribute) → list       procedure
```

These are procedures on SXML attribute forms. `sxml-attribute?` is a predicate for SXML attribute forms. It checks if *object* is a list whose first element is the symbol `@`. `sxml-attribute-attributes` returns the list of name-value-lists of the attributes form. Both procedures are exported by the `surflets/sxml` structure.

The translator translates list elements which are numbers and symbols to their string representation (except for the first element, of course). She scans strings for the special characters `&`, `"`, `>` and `<` and replaces them by their HTML equivalents, and she ignores `#f` and the empty list. See below the special SXML tag `plain-html` to see how to insert HTML code untranslated. Furthermore, the translator accepts `input-fields` as list elements, which are translated to their HTML representation. See below for details on input fields.

Using lists to represent HTML allows the programmer to define operations on it. Most programmers construct their lists dynamically, often by using `quasiquote` (the symbol `'`) and `unquote` (the symbol `,`). *E.g.*

```
'(html (title ,my-title)
      (body (p "Hello, " ,(get-user-full-name))))
```

See below for how to create your own SXML.

### Special SXML tags

The SXML to HTML translator accepts some special SXML tags that don't directly translate to an HTML tag.

```
(url URL [text])           SXML-tag
```

Inserts a link to *URL*, named with *text*. *text* defaults to *URL*. Takes at least one argument. *E.g.*

```
(url "/" "Main menu") ⇒ <a href="/">Main menu</a>
(url "go.html")       ⇒ <a href="go.html">go.html</a>
```

*Oops:* `url` does not accept extra attributes for the `'A'` tag of HTML. This should be fixed in a future version.

<code>(nbsp)</code>	SXML-tag
Inserts the HTML sequence "&nbsp;". Takes no arguments.	
<code>(plain-html html ...)</code>	SXML-tag
Inserts <i>html</i> without any changes, thus it works like a quote. Takes any number of arguments.	
<code>(*COMMENT* comment ...)</code>	SXML-tag
Inserts a comment, <i>i.e.</i> <i>comment</i> enclosed between <code>&lt;!--</code> and <code>--&gt;</code> . Takes any number of arguments.	
<code>(surfilet-form k-url [method] [attributes] [SXML ...])</code>	SXML-tag
Inserts HTML code for a web form. See below for details. <i>k-url</i> usually is a continuation-URL. <i>method</i> is the method to be used by the client to transfer the webform data. Possible values are the symbols GET, get, POST, post, the first two specifying the GET method, the last two the POST method. <i>method</i> defaults to the GET method. <i>attributes</i> are attributes for the created web form, <i>e.g.</i> <code>(@ (enc-type "text/plain"))</code> . The remaining arguments are taken as SXML and translated as usually. Takes at least one argument. Note that the attributes form may come at position three.	

### Do it yourself: your own SXML

The `send-html` procedures use a standard set of translation rules to translate from SXML to HTML. However, you may define your own set of translation rules or extend the given ones as you see fit. For this, a short introduction to the translation process.

The translation process takes place in two steps. Step one translates the given SXML to low level SXML, essentially a rough form of HTML in list notation. Step two takes this low level SXML and prints it to a port. Step one is performed by `sxml->low-level-sxml`, step two by `display-low-level-sxml`. All procedures and rules presented in this subsection are exported from `surfilets/sxml`.

<code>(sxml-&gt;low-level-sxml sxml rules)</code>	$\longrightarrow$	<code>low-level-sxml</code>	procedure
Takes an SXML object (which is essentially a list) and a list of SXML rules (more on this below) and translates it to low level SXML. This procedure is an alias to the <code>pre-post-order</code> procedure of Oleg Kiselyov's SSAX module. It is an error if no rule triggers (see below for when a rule triggers). However, it is no error if multiple rules trigger; the first rule in the <i>rules</i> list wins.			

(display-low-level-sxml *low-level-sxml port*)  $\longrightarrow$  *boolean* procedure

Takes low level SXML and displays it to a port. She traverses the list *low-level-sxml* depth-first, ignores the empty list and #f, executes thunks and displays all other elements, usually strings and characters, to *port*. Returns #t if she wrote anything, #f otherwise. This function is basically the SRV:send-reply procedure of Oleg Kiselyov's SSAX module.

(sxml->string *sxml rules*)  $\longrightarrow$  *string* procedure

Combines step one and two of the translation process and returns the resulting string, *i.e.* it calls display-low-level-sxml with the result of a call to sxml->low-level-sxml and a string port, returning the content of the string port.

An SXML-rule consists of a *trigger*, which is a symbol, and the *handler*, which is a translation procedure.

There are three types of rules, each of which is a dotted list:

(⟨*trigger*⟩ \*preorder\* . ⟨*handler*⟩)

When sxml->low-level-sxml sees the ⟨*trigger*⟩ as the first element of a list, she calls ⟨*handler*⟩ with the *whole* list as arguments and replaces the list with the result of that call (which must be a single value). Note that the arity of the handler determines how many elements the list with the trigger may or must contain.

(⟨*trigger*⟩ . ⟨*handler*⟩)

When sxml->low-level-sxml sees the ⟨*trigger*⟩ as the first element of a list, she calls herself on the remaining elements of the list and then calls the ⟨*handler*⟩ with the trigger and the results of those calls as arguments.

(⟨*trigger*⟩ ⟨*new-rules*⟩ . ⟨*handler*⟩)

When sxml->low-level-sxml sees the ⟨*trigger*⟩ as the first element of a list, she temporarily prepends ⟨*new-rules*⟩ to the current rule set while calling herself on the remaining elements of the list. She then calls the ⟨*handler*⟩ with the trigger and the results of those calls as arguments. As the new rules are prepended, this rule allows the temporary override of some rules.

There are two special triggers, who may trigger for all elements of the SXML, not only the first element of a list:

- \*text\* triggers for atoms in the SXML list, *i.e.* usually strings and characters. The handler is called with the symbol \*text\* and the atom as arguments.

- `*default*` triggers whenever no rule triggered, including `*text*`. If called for a list whose first element did not trigger a rule, the handler is called with the whole list. If called for an atom, the handler is called with the symbol `*text*` and the atom as arguments.

The `surflets/sxml` structure defines some basic rules:

<code>default-rule</code>	SXML-rule
<code>text-rule</code>	SXML-rule
<code>attribute-rule</code>	SXML-rule

These are the three basic rules exported by the `surflets/sxml` structure. `default-rule` creates the leading and trailing HTML tag and encloses the attributes. `text-rule` just inserts the given text with the special HTML characters `&`, `"`, `>` and `<` escaped. `attribute-rule` triggers for the attributes form and creates attributes like `selected` or `color="red"`.

The `surflets/surflet-sxml` add the rules for the special SXML tags to this list:

<code>url-rule</code>	SXML-rule
<code>nbsp-rule</code>	SXML-rule
<code>plain-html-rule</code>	SXML-rule
<code>comment-rule</code>	SXML-rule
<code>surflet-form-rule</code>	SXML-rule

These are the rules for the special SXML tags mentioned above, namely `url`, `nbsp`, `plain-html`, `*COMMENT*` and `surflet-form`.

<code>default-rules</code>	list
<code>surflet-sxml-rules</code>	list

These are rule sets. `default-rule` contains the rules `default-rule`, `attribute-rule`, `text-rule`, `comment-rule`, `url-rule`, `plain-html-rule` and `nbsp-rule`. `surflet-sxml-rules` extends this list by `surflet-form-rule` and a rule for input fields.

`(surflet-sxml->low-level-sxml sxml) → low-level-sxml` procedure

This uses the `surflet-sxml-rules` to translate `sxml` to low level SXML, performin step one of the translation process.

## 6.2.9 Continuation-URL

The continuation-URL represents the point in the computation of a session of a SURflet where the computation was halted by the SURflet handler. When a

browser requests a continuation-URL, the SURflet handler looks up the continuation in its tables and reifies it, allowing the session of the SURflet to resume its computation.

The procedures to access the continuation-URL are the following. They are exported by the `surflet-handler/resume-url` structure. Sorry for the double naming `resume-url` and `continuation-URL`.

```
(resume-url? string)  → boolean           procedure
(resume-url-ids resume-url) → session-id continuation-id  procedure
(resume-url-session-id resume-url) → session-id           procedure
(resume-url-continuation-id resume-url) → continuation-id procedure
```

These inspect values of a resume url. `resume-url?` is a predicate for resume urls (the same as continuation urls). Note that it only operates on strings. `resume-url-ids` returns the session- and the continuation-id that is stored in the *resume-url*. `resume-url-session-id` and `resume-url-continuation-id` return only the session- or the continuation-id, respectively.

### 6.2.10 Input fields

The SURflets support all input fields defined for HTML 2.0 and allow the creation of own input fields. `input-fields` are first order values, that represent the actual input field of the web page in the SURflet. For that, this documentation distinguishes the *browser* value from the *Scheme* value of an input field. The browser value is the string representation of the input field data the browser sends. The Scheme value is the value in the SURflet the input field reports as its value, which may be of any type, not only strings.

Here is a short overview on how to use input fields. See also the *howto* for more informations. First, you create the `input-field` that represents the input field you want to use. Then you put this `input-field` into the SXML of the web page at the place the input field shall appear. After `send-html/suspend` has returned with the next `surflet-request`, you call `get-bindings` with that `surflet-request` and collect the resulting bindings. Last, you call `input-field-value` (or `raw-input-field-value`) with your `input-field` and the collected bindings to get the Scheme representation of the value the user has entered. Here is a small example:

```
(define-structure surflets surflet-interface
  (open scheme-with-scssh
    surflets)
  (begin
    (define (main req)
```



```

(let* ((text-input (make-text-field))
      (req (send-html/suspend
             (lambda (k-url)
               ' (html
                 (body
                  (surflet-form
                   ,k-url
                   (p "Enter some asd text: " ,text-input)
                   ,(make-submit-button)))))))
      (bindings (get-bindings req))
      (text (input-field-value text-input bindings)))
(send-html/finish
 ' (html
   (body
    (p "You've entered '" ,text "'."))))))

```

**Getting the bindings** The `surflets/bindings` structures exports the necessary functions to create bindings and extract values from them:

`(get-bindings surflet-request)`  $\longrightarrow$  *bindings* procedure

This returns an association list representing the data the browser has sent, usually the content of a webform. The name of the input fields are the keys, their browser values the values. The values are already unescaped. `get-bindings` can (currently) only handle `application/x-www-form-urlencoded` data. You can call `get-bindings` on both GET and POST requests, even multiple times (even on POST requests).

`(extract-bindings key bindings)`  $\longrightarrow$  *list* procedure

`(extract-single-binding key bindings)`  $\longrightarrow$  *string* procedure

These extract values from the *bindings* as returned by `get-bindings`. *key* may be a string or a symbol which will be translated to a string before use. `extract-bindings` returns a list of all values from *bindings* whose key is *key*. `extract-single-binding` returns the value from the binding whose key is *key* and raises an error if there is more than one such binding. The two procedures are the same as in PLT's webserver.

`get-bindings` must access the "Content-length" header field to handle POST requests. `surflets/bindings` also exports the procedure that does that job:

`(get-content-length headers)`  $\longrightarrow$  *number* procedure

Returns the value of the “Content-length” header as a number, as present in *headers*, e.g. from `surflet-request-headers`. Will raise an error if there is no “Content-length” header or the header is illformed, e.g. contains no number.

**Retrieving the Scheme values** The `surflets/input-field-value` structure provides the functions necessary to retrieve the Scheme value of input fields.

`(raw-input-field-value input-field bindings)`  $\longrightarrow$  *any type* procedure  
`(input-field-value input-field bindings [error-value])`  $\longrightarrow$  *any type* procedure

These extract the Scheme value of an *input-field*, given the *bindings* of the last request. Asking for a Scheme value may raise an error. Some error conditions are: the input field was not present in the bindings, the transformer could not generate a Scheme value for the browser value, or some other error occurred in a maybe malfunctioning transformer. In any case, `raw-input-field-value` won’t catch that error, while `input-field-value` will catch it and provide *error-value* as the *input-field*’s Scheme value, which defaults to `#f`.

`(input-field-binding input-field bindings)`  $\longrightarrow$  *binding* procedure

This returns the first binding in *bindings* that belongs to the given *input-field* (i.e. has *input-field*’s name as key).

**Creating and using input fields** The procedures for the creation of the input fields mentioned in HTML 2.0 are the following. They are exported by the `surflets/surflet-input-fields`. Note that most of the time, you may omit any of the optional arguments, e.g. you may only specify some further attributes to `make-text-field` without specifying a default value. Keep in mind that `input-field-value` catches the error that may occur if an *input-field* is asked for its Scheme value and may return any (previously chosen) value instead.

`(make-text-field [default] [attributes])`  $\longrightarrow$  *input-field* procedure  
`(make-number-field [value] [attributes])`  $\longrightarrow$  *input-field* procedure  
`(make-password-field [default] [attributes])`  $\longrightarrow$  *input-field* procedure  
`(make-textarea [default] [rows] [columns] [readonly?] [attributes])`  $\longrightarrow$  *input-field* procedure

These create various input field where the user types something in. *default* is the text or the number that the browser initially displays in the input field. *attributes* are some further attributes for the input field in SXML notation—it needs to be a list of the following form:

`(@ (tag value)`  
`...)`

`make-text-field` creates a regular text input field. Its Scheme value is a string. `make-number-field` creates a regular text input field, whose Scheme value is a number. It is an error if the input field does not contain a number. `make-password-field` creates a text input field that will display stars instead of the typed text. Its Scheme value is a string. `make-textarea` creates a possibly multi line text input field. *rows* specifies how many rows of the text the browser will display at once and defaults to 5. *columns* specifies how many columns the browser will display at once and defaults to 20. Note that if you only supply one number, it will be interpreted as the *rows* argument. *readonly?* is a boolean that tells the browser whether to disallow changes of the displayed text.

`(make-hidden-input-field [default] [attributes])`  $\longrightarrow$  *input-field* procedure

Creates a hidden input field, *i.e.* a input field that the browser won't display but whose value the browser will send. This input field is provided for completeness; you usually won't need it, as all values in your SUrllet will survive the emission of a web page. *default* is this value the browser will send. Note that although the argument is marked as optional you usually want to provide it. *attributes* are some further attributes for the input field in SXML notation.

<code>(set-text-field-value! <i>input-field</i>)</code>	$\longrightarrow$ <i>undefined</i>	procedure
<code>(set-number-field-value! <i>input-field</i>)</code>	$\longrightarrow$ <i>undefined</i>	procedure
<code>(set-hidden-field-value! <i>input-field</i>)</code>	$\longrightarrow$ <i>undefined</i>	procedure
<code>(set-password-field-value! <i>input-field</i>)</code>	$\longrightarrow$ <i>undefined</i>	procedure
<code>(set-textarea-value! <i>input-field</i>)</code>	$\longrightarrow$ <i>undefined</i>	procedure

These set the default value of the according input field after it has been created. Although the procedure may not complain, it is an error, if *input-field* is not the expected type of *input-field*, *e.g.* if the argument to `set-text-field-value` was not created by `make-text-field`.

<code>(make-submit-button [caption] [attributes])</code>	$\longrightarrow$ <i>input-field</i>	procedure
<code>(make-reset-button [caption] [attributes])</code>	$\longrightarrow$ <i>input-field</i>	procedure
<code>(make-image-button <i>image-source</i> [attributes])</code>	$\longrightarrow$ <i>input-field</i>	procedure

These create buttons on the web page which the user can click on. *caption* is the text that is displayed on the button. If not specified, the browser will choose a text, usually depending on the local language setting on the browser side. *attributes* are some further attributes for the input field in SXML notation. `make-submit-button` creates the regular button to submit the web form data. As HTML 2.0 specifies that the value of a submit button is its caption, its Scheme value is its caption, too. `make-reset-button` creates the button to reset all input fields of the web form to their default values. As the browser does not send data for reset

buttons, it does not have a Scheme value, *i.e.* asking for a value will raise an error. `make-image-button` creates a picture button. Its Scheme value is a pair indicating the x- and y-coordinates of the picture where the user has clicked to. The argument *image-source* is not optional and is the string URL of the displayed picture.

```
(make-checkbox [checked?] [attributes]) → input-field      procedure
(make-annotated-checkbox value [checked?] [attributes]) → input-field procedure
```

These create checkboxes. *checked?* says whether the browser should initially mark the checkbox as checked. *attributes* are some further attributes for the input field in SXML notation. If it was checked the Scheme value of a checkbox made by `make-checkbox` is `#t`. If it was checked, the Scheme value of a checkbox made by `make-annotated-checkbox` is its *value* provided during its creation where *value* may be chosen arbitrarily. Note that HTML 2.0 specifies that browsers should not send data for unmarked checkboxes, thus asking for the Scheme value of an unmarked checkbox will raise an error. It is recommended to use `input-field-value` to ask for the Scheme value of a checkbox. This will catch the error and will instead return `#f` by default.

```
(check-checkbox! input-field) → undefined      procedure
(uncheck-checkbox! input-field) → undefined    procedure
(set-checkbox-checked?! input-field checked?) → undefined procedure
```

These change the *checked?* field of a checkbox that tells the browser whether it should initially mark the checkbox as checked. `check-checkbox!` tells the browser to do so, `uncheck-checkbox!` does not tell the browser to do so, and `set-checkbox-checked?!`  does so depending on *checked?*. It is an error if *input-field* was not created by `make-checkbox` or `make-annotated-checkbox`.

```
(make-radio-group) → procedure                 procedure
(make-annotated-radio-group) → procedure       procedure
```

These return generators for radio buttons. Radio buttons usually are part of a group of radio buttons of which only one may be selected at any time. The procedures return a procedure that creates radio button input-fields that belong to the same group.

The returned procedures accept a *value* argument, an optional *checked?* argument and an optional *attributes* argument. They return an `input-field`, the actual radio button. For `make-radio-group`, *value* must be a string, for `make-annotated-radio-group`, *value* may be any Scheme value. The Scheme value of any member of the group of radio buttons is the *value* of the marked radio button that was provided during its creation. *checked?* determines whether the browser will initially mark the

radio button. Note that you are able to tell the browser to initially mark more than one radio button, but in which case the browser's behavior is undefined. *attributes* are some further attributes for the input field in SXML notation.

```
(check-radio! input-field) → undefined      procedure
(uncheck-radio! input-field) → undefined    procedure
(set-radio-checked?! input-field checked?) → undefined  procedure
```

These change the *checked?* field of a radio button that tells the browser whether it should initially mark the radio button as checked. *check-radio!* tells the browser to do so, *uncheck-radio!* does not tell the browser to do so, and *set-radio-checked?!* does so depending on *checked?*. It is an error if *input-field* was not created by the procedures returned by *make-radio-group* or *make-annotated-radio-group*.

```
(make-select select-options [multiple?][attributes]) → input-field  procedure
```

This creates a select boxes. Other names are “drop down menu” or simply “list”. *select-options* is either a list of *select-options* created with the procedures presented below or a list of strings. In the latter case the strings are automatically translated into *select-options*. *multiple?* allows multiple selections in the select box. *attributes* are some further attributes for the input field in SXML notation. Note that you will only get multiple Scheme values for a select box that allows multiple selections, if you specify the *multiple?* argument; providing the according attribute in *attributes* won't work (you will get the value of the first selection only).

```
(make-simple-select-option tag [selected?][attributes]) → select-option  procedure
(make-annotated-select-option tag value [selected?][attributes]) → select-option  procedure
```

These create the options for a select box, to be used as arguments to *make-select*. *tag* is a string that will be displayed as an option of a select box. *value* is an arbitrary Scheme value that will be the Scheme value of the select input field that contains the option. For simple select options this is the same as *tag*. *selected?* determines whether the browser should preselect the option. *attributes* are some further attributes for the input field in SXML notation.

```
(select-option? object) → boolean      procedure
```

This is a predicate for *select-options*.

```
(select-select-option! tag input-field) → undefined      procedure
(unselect-select-option! tag input-field) → undefined    procedure
(set-select-option-selected?! tag input-field selected?) → undefined  procedure
```

These change the `selected?` field of a select option that tells the browser to preselect it. `select-select-option!` tells the browser to preselect it, `unselect-select-option!` does not tell it to do so and `set-select-option-selected!` does so depending on *selected?*. Note that you access the select option by providing the *tag* and the select *input-field* in which the select option is saved. *tag* is either the tag of the select option or an index with 0 being the first select option of that select input field. However, the change will affect all select input fields that use the same select option. If there are different select options with the same tag in a select input field, the procedures will only touch one of them.

*Oops:* Unfortunately, the order of the arguments (index, object) is the opposite of what is usual in Scheme (object, index). This should be fixed in a future version.

```
(add-select-option! input-field select-option) → undefined procedure
(delete-select-option! input-field select-option) → undefined procedure
```

These add or remove *select-option* to or from the select *input-field*, respectively.

### Do it yourself: your own input fields

The SURflets library allows the creation of arbitrary own input fields. The relevant procedures are exported by the `surflets/my-input-fields` structure.

```
(make-input-field name type transformer attributes html-tree-maker) → input-field procedure
(make-multi-input-field name type transformer attributes html-tree-maker) → input-field procedure
```

These are the two constructors for input-fields.

*name* is the name of the input field as used in the HTML. You have to make sure that this name is unique across your web page, e.g. by using `generate-input-field-name` presented below.

*type* is the type of the input field and mainly meant as a label for debugging. You may choose an arbitrary value for it.

*transformer* is a procedure that accepts the created `input-field` and some other value as arguments and returns the (single) Scheme value of the input field. For `make-input-field`, the other value is the string representation of the value the user has entered in the represented input field, as sent by the browser. For `make-multi-input-field`, the other value is an association list of all data the browser has sent, the names being the key and the entered data being the value. This is the very same list as returned by `get-bindings`, see above. When the *transformer* cannot create a Scheme value for the `input-field`, she should raise an error.

*attributes* takes some extra information you want to store along with the *input-field*. You may choose an arbitrary value for it.

*html-tree-maker* is a procedure that takes the created *input-field* as argument and returns its representation in SXML.

(generate-input-field-name *prefix*)  $\longrightarrow$  *string* procedure

This generates a pseudo unique name based on prefix. Subsequent calls with the same prefix are guaranteed to never return the same string.<sup>5</sup>

(input-field-name <i>input-field</i> )	$\longrightarrow$ <i>string</i>	procedure
(input-field-type <i>input-field</i> )	$\longrightarrow$ <i>any type</i>	procedure
(input-field-transformer <i>input-field</i> )	$\longrightarrow$ <i>procedure</i>	procedure
(input-field-attributes <i>input-field</i> )	$\longrightarrow$ <i>any type</i>	procedure
(input-field-html-tree-maker <i>input-field</i> )	$\longrightarrow$ <i>procedure</i>	procedure
(input-field-html-tree <i>input-field</i> )	$\longrightarrow$ <i>sxml</i>	procedure
(input-field-multi? <i>input-field</i> )	$\longrightarrow$ <i>boolean</i>	procedure

These inspect input field values. *input-field-name* returns the name of the input field as used in its HTML representation. *input-field-type* returns a string indicating the type of the input field, *e.g.* "radio" or "text". For individual input fields it may return a value of any type. *input-field-transformer* returns the transformer procedure that is used to transform the browser value of the input field to a Scheme value. *input-field-attributes* returns the attributes that were stored along with the input field. *input-field-html-tree-maker* returns the procedure that creates the SXML representation of the input field. *input-field-html-tree* returns the SXML representation of the input field. *input-field-multi?* returns #t if the input field was created with *make-multi-input-field*, #f otherwise. The transformer of a multi-input-field gets the browser bindings as second argument while the transformer of a normal (non-multi) input-field gets the string representation of the entered data as second argument.

(set-input-field-attributes! *input-field new-attributes*)  $\longrightarrow$  *undefined* procedure

This allows the mutation of the attributes of the *input-field* to *new-attributes*.

(touch-input-field! *input-field*)  $\longrightarrow$  *undefined* procedure

This forces the recalculation of the SXML representation of the *input-field* using its *html-tree-maker* procedure.

---

<sup>5</sup>Well, never say never: if the structure is reloaded, the counter is reset and *generate-input-field-name* will return the same names again.

### 6.2.11 Web addresses

The SURflets library allow you to determine which link or button a user used to leave a page. The links are called evaluable web addresses. The `surflets/returned-via` structure provides procedures and syntax for this.

`(returned-via return-object bindings)`  $\longrightarrow$  *any value*                      procedure  
`(returned-via? return-object bindings)`  $\longrightarrow$  *any value*                      procedure

Determines, whether the user left the web page using *return-object*. *bindings* are the bindings as returned by `get-bindings`. If *return-object* is an `input-field`, `returned-via` returns its Scheme value as reported by `input-field-value`. The input field usually can only be a submit or an image button. If *return-object* is not an `input-field`, `returned-via` assumes it is an evaluable web address. If the user did not use the evaluable web address to leave the web page, `returned-via` returns `#f`. Otherwise, when the evaluable web address is annotated, `returned-via` returns its annotation, otherwise just `#t`. `returned-via?` is an alias for `returned-via`. The type of the return value depends on the type of *return-object*.

`(case-returned-via <key> <clause> ...)`                      syntax

This works like `case` with some flavor of `cond`. Instead of `eq?` it uses `returned-via` to determine which *<clause>* applies. *<key>* is the *bindings* argument to `returned-via` (see above for the arguments of `returned-via`).

A clause is of the form

$((\langle datum \rangle \dots) \langle expression \rangle \dots)$

where each *<datum>* is the *return-object* argument of `returned-via`. If for any of the *<datum>* `returned-via` returns a true value, the *<expression>*s are evaluated.

Alternatively, a clause may be of the form

$((\langle datum \rangle \dots) \Rightarrow \langle procedure \rangle \dots)$

If for any of the *<datum>* `returned-via` returns a true value, *<proc>* is called with that value.

The last possible clause is an "else" clause of the form

$(\text{else } \langle expression \rangle \dots)$

which applies when the previous clauses don't apply.

`case-returned-via` returns the value(s) of the *<expression>* that was evaluated last.



## Evaluatable web addresses

The `surfleets/addresses` structure provides procedures to create evaluatable web addresses. Evaluatable web addresses are used just like web addresses with the difference that `returned-via` can tell whether the user used this web address to leave the web page.

`(make-address)`  $\longrightarrow$  *address-procedure* procedure

This creates an evaluatable web address. *address-procedure* is a procedure that accepts messages. If the message is a string, *address-procedure* will assume it is a continuation URL and will return a web address that can be used as a link. If the message is the symbol `address`, *address-procedure* will return the real address object.

`(make-annotated-address)`  $\longrightarrow$  *address-procedure* procedure

This creates an annotated evaluatable web address. *address-procedure* is a procedure that accepts messages. The procedure accepts either a string and an optional annotation which may be any Scheme value, or it accepts only the symbol `address`. In the first case, it will assume the string is a continuation URL and will return a web address that can be used as a link. In the latter case, it will return the real address object.

*Oops:* Evaluatable web address cannot be used as the action URL of web forms.

`(address-name address)`  $\longrightarrow$  *string* procedure

`(address-annotated? address)`  $\longrightarrow$  *boolean* procedure

`(address-annotation address)`  $\longrightarrow$  *any type* procedure

These inspect real address objects as returned by the evaluatable web addresses when given the symbol `address`. `address-name` returns the name of the *address* as used in the browser data. `address-annotated?` indicates whether *address* is annotated. `address-annotation` returns the annotation of *address*. If *address* is not annotated, it returns `#f`.

### 6.2.12 Callbacks

The `SURfleets` library allows to add a callback to a link. When the user of a web page clicks on the link, the callback will be executed. `send-html/suspend` (usually) won't return in that case.

`(make-callback callback-procedure)`  $\longrightarrow$  *continuation-URL* procedure

This creates a callback. When a user clicks on a link to the continuation URL `make-callback` has returned, *callback-procedure* will be called with the according `surflet-request`. *callback-procedure* should not return.

`make-callback` works with continuations. Therefore, it is not sensible to create callbacks on toplevel, nor is it sensible to reuse callbacks. Instead, create your callback every time and right before you need it.

If *callback-procedure* returns, `make-callback` will return *again*, this time with the value returned by *callback-procedure*. Note that in this case the continuation that was active at the time of the call to `make-callback` is restored. Or, in short, don't let *callback-procedure* return if you want to avoid headaches.

`(make-annotated-callback callback-procedure) → procedure procedure`

This creates a callback generator. The returned procedure accepts any number of arguments *args* and returns a continuation URL. When the user clicks on a link to the continuation URL, *callback-procedure* will be called with the arguments *args* previously provided.

It is an error, if *callback-procedure* returns. You should create a fresh annotated callback every time and right before you need it, as the continuation that was active at the time of the call to `make-annotated-callback` is restored.

`callback-function` `procedure`

Use this procedure as the *callback-procedure* argument to `make-annotated-callback` to call arbitrary procedures with arbitrary arguments.

Here are some examples. The first example shows how you can use an annotated callback. Note that it does not need to use `send-html/suspend`.

```
(define-structure surflet surflet-interface
  (open surflets
    surflets/callbacks
    scheme-with-scssh)
  (begin

    (define (main req)
      (let ((language (make-annotated-callback result-page)))
        (send-html
          '(html
            (head (title "Multi-lingual"))
            (body
```

```

(h2 "Select your language:")



  (li (url ,(language "Hello, how are you?")
          "English")
      (li (url ,(language "Hallo, wie geht es Ihnen?")
              "Deutsch")))))))

(define (result-page req text)
  (send-html/finish
    ' (html
      (head (title "Greeting"))
      (body
        (h2 ,text)))))

))

```

Replacing the main procedure with the following definition will have the same result:

```

(define (main req)
  (let ((language (make-annotated-callback callback-function)))
    (send-html
      ' (html
        (head (title "Multi-lingual"))
        (body
          (h2 "Select your language:")
          (ul
            (li (url ,(language result-page "Hello, how are you?")
                    "English")
              (li (url ,(language result-page "Hallo, wie geht es Ihnen?")
                      "Deutsch")))))))
    ))

```

### 6.2.13 Outdater

The SURflets library allows the user to navigate through the web pages back and forth as she sees fit. However, sometimes you want to make sure, that a submission is done only once. For this, the SURflets provide outdater objects that take care of this.

`(make-outdater)`  $\longrightarrow$  *outdater* procedure

Creates an outdater object.

`(if-outdated <outdater> <consequence> <alternative>)` syntax

Using the  $\langle outdater \rangle$ , this makes sure, the  $\langle alternative \rangle$  is executed at most once, *i.e.* the first time the  $\langle outdater \rangle$  is used in such a form, the  $\langle alternative \rangle$  is evaluated. Every subsequent evaluation of the *if-outdated* form with the  $\langle outdater \rangle$  will evaluate the  $\langle consequence \rangle$ , usually something similar to what *show-outdated* does.

`(show-outdated url)`  $\longrightarrow$  *no return value* procedure

Emits a regular web page to the client informing the user (in English) that “the page or action you requested relies on outdated data”. It offers a “reload” link that points to *url* to get current data. Usually, *url* is a call-back URL the calls the according procedure. See the admin SURflets for examples, *e.g.* `scheme/web-server/root/surlfets/admin-surflet.scm`.

#### 6.2.14 Simple SURflets

PLT offers an API to create simple servlets (which are their analogues to our SURflets). The `simple-surflet-api` structure offers the procedures with the same name as in the PLT API. With that, SURflets can look as simple as this (`scheme/web-server/root/surlfets/add-simple.scm`, see also `simple-surflet.scm` in the same directory for a larger example:

```
(define-structure surflet surflet-interface
  (open scheme-with-scsh
    surflets
    simple-surflet-api
  )
  (begin

    (define (main req)
      (let* ((number-1 (single-query (make-number "First number:")))
            (number-2 (single-query (make-number "Second number:"))))
        (inform (format #f "~a + ~a = ~a"
                        number-1
                        number-2
                        (+ number-1 number-2))))
      (final-page "Session finished.)))

  ))
```

The procedures are the following.

`(single-query query)`  $\longrightarrow$  *any type* procedure

Asks the user one single questions based on *query* and returns her answer.

(queries *queries*)  $\longrightarrow$  *list* procedure

Asks the user multiple questions based on the list of *queries* and returns her answers in a list.

(form-query *named-queries*)  $\longrightarrow$  *list* procedure

Asks the user multiple queries based on the list of *named-queries* and returns her answers in a pseudo association list. *named-queries* is a list of two element lists. The first element of those lists is a symbol identifying the query, the second is the query. The resulting pseudo association list contains two element lists, where the first element is the symbol and the second element the user's answer to the query. The result can be read using the `extract/single` and `extract` procedures.

(inform *title* [*text* ...])  $\longrightarrow$  *surflet-request* procedure

Sends a web page title *title* with the *text* to the user as an information. The returned `surflet-request` is usually discarded. Takes at least one argument.

(final-page *title* [*text* ...])  $\longrightarrow$  *no return value* procedure

This sends the last page of the session to the user, titled *title* and containing *text*. This is the analog to `send/finish`. Takes at least one argument.

(make-text *invitation*)  $\longrightarrow$  *query* procedure

(make-number *invitation*)  $\longrightarrow$  *query* procedure

(make-password *text*)  $\longrightarrow$  *query* procedure

(make-boolean *invitation*)  $\longrightarrow$  *query* procedure

(make-radio *invitation choices*)  $\longrightarrow$  *query* procedure

(make-yes-no *invitation yes-text no-text*)  $\longrightarrow$  *query* procedure

These create the various queries. *invitation* is a text displayed in front of the input field, e.g. "Please enter your password:". `make-text` creates a text input field, `make-number` creates a number input field (i.e. a text input field that only accepts numbers as inputs), `make-password` creates a password input field, `make-boolean` creates a checkbox, `make-radio` creates a group of radio buttons of which only one can be selected and `make-yes-no` creates a radio group that allows the choices *yes-text* and *no-text*.

The value of `make-text`, `make-number` and `make-password` is the text or number entered into the input field. The value of `make-boolean` is `#t` or `#f`. The value of `make-radio` and `make-yes-no` is the selected choice, a string.

(extract/single <i>symbol table</i> )	→	<i>any value</i>	procedure
(extract <i>symbol table</i> )	→	<i>list</i>	procedure

Return the answer of a user to a query. *table* is the result of form-query, *symbol* the symbol used to identify the query of interest. For extract/single, it is an error if there is more than one query in *table* that is identified by *symbol*.

## Chapter 7

# 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 wuftp. 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. Anonymous 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

## Chapter 8

# FTP Client

The `ftp` structure lets you transfer files between networked machines from the Scheme Shell, using the File Transfer Protocol as described in RFC 959.

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.

`(ftp-connect host login password passive? [log-port])`  $\longrightarrow$  *connection*    procedure

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.

`(ftp-type <name>)`  $\longrightarrow$  *ftp-type*    syntax  
`(ftp-set-type! connection ftp-type)`  $\longrightarrow$  *undefined*    procedure

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.

`(ftp-rename connection old new)`  $\longrightarrow$  *undefined*    procedure

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 conneciton 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 9

# 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 ondrive.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 10

# 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.<sup>1</sup>

*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

---

<sup>1</sup>In 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 11

# 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.

### 11.1 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`.

### 11.2 Time

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

`(rfc868-time/tcp host)`  $\longrightarrow$  *string* procedure  
`(rfc868-time/udp host [timeout-or-#f])`  $\longrightarrow$  *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 12

# 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.

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

These transactions represent the commands of the SMTP protocol for use in smtp-transaction and smtp-transaction/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 13

# 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 10.) 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 14

# DNS Client Library

**Used files:** `dns.scm`

**Name of the package:** `dns`

### 14.1 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

### 14.2 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) → 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.

### 14.3 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!*)  $\rightarrow$  *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 14.5). 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?]*)  $\rightarrow$  *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?]*)  $\rightarrow$  *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?]*)  $\rightarrow$  *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?]*)  $\rightarrow$  *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?]*)  $\rightarrow$  *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?]*)  $\rightarrow$  *IP-address or #f* procedure

(*maybe-dns-lookup-ip IP-string/IP-address [nameserver list][use-cache?]*)  $\rightarrow$  *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?]*)  $\longrightarrow$  *FQDN* procedure  
 (*system-fqdn [nameserver list][use-cache?]*)  $\longrightarrow$  *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.

(*dns-check-nameservers [nameserver list]*)  $\longrightarrow$  *undefined* procedure

*dns-check-nameservers* checks if the given nameservers are reachable. If no argument is given, the nameservers in */etc/resolv.conf* are checked. Information about the status of the nameservers is printed to the current output port.

## 14.4 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?]*)  $\longrightarrow$  *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*)  $\longrightarrow$  *network-protocol* syntax  
 (*network-protocol? thing*)  $\longrightarrow$  *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?]*)  $\longrightarrow$  *dns-message* procedure

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

(dns-message? <i>thing</i> )	→ <i>boolean</i>	procedure
(dns-message-query <i>dns-message</i> )	→ <i>message</i>	procedure
(dns-message-reply <i>dns-message</i> )	→ <i>message</i>	procedure
(dns-message-cache? <i>dns-message</i> )	→ <i>boolean</i>	procedure
(dns-message-protocol <i>dns-message</i> )	→ <i>protocol</i>	procedure
(dns-message-tried-nameservers <i>dns-message</i> )	→	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*]) → *undefined* procedure

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

(message? <i>thing</i> )	→ <i>boolean</i>	procedure
(message-header <i>message</i> )	→ <i>header</i>	procedure
(message-questions <i>message</i> )	→ <i>question list</i>	procedure
(message-answers <i>message</i> )	→ <i>resource-record list</i>	procedure
(message-nameservers <i>message</i> )	→ <i>resource-record list</i>	procedure
(message-additionals <i>message</i> )	→ <i>resource-record list</i>	procedure
(message-source <i>message</i> )	→ <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*]) → *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*) → *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> )	→ <i>boolean</i>	procedure
(header-id <i>header</i> )	→ <i>number</i>	procedure
(header-flags <i>header</i> )	→ <i>flags</i>	procedure
(header-question-count <i>header</i> )	→ <i>number</i>	procedure

(header-answer-count <i>header</i> )	→ <i>number</i>	procedure
(header-nameserver-count <i>header</i> )	→ <i>number</i>	procedure
(header-additional-count <i>header</i> )	→ <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-zero <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 if 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 <i>message-type</i> )	→ <i>symbol</i>	procedure
(message-type-number <i>message-type</i> )	→ <i>number</i>	procedure

message-type constructs a member of the enumeration from name  $\langle type-name \rangle$  listed in Table 14.1. 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.

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

Table 14.1: Message types

(resource-record? <i>thing</i> )	→ <i>boolean</i>	procedure
(resource-record-name <i>resource-record</i> )	→ <i>string</i>	procedure
(resource-record-type <i>resource-record</i> )	→ <i>message-type</i>	procedure
(resource-record-class <i>resource-record</i> )	→ <i>message-class</i>	procedure
(resource-record-ttl <i>resource-record</i> )	→ <i>number</i>	procedure
(resource-record-data <i>resource-record</i> )	→ <i>resource-record-data-...</i>	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-name *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.

## 14.5 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.

## 14.6 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-&gt;ip-string IP-address)</code>	$\longrightarrow$	<code>ip-string</code>	procedure
<code>(ip-string-&gt;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

\*COMMENT\*, 69

address-annotated?, 81  
address-annotation, 81  
address-name, 81  
address32->ip-string, 113  
alist-path-dispatcher, 15  
attribute-rule, 71

bad-address, 105  
bad-address?, 105  
bad-nameserver, 105  
bad-nameserver?, 105

cache-answer, 111  
cache-time, 111  
cache-ttl, 111  
cache?, 111  
callback-function, 82  
case-returned-via, 80  
cgi-form-query, 30  
cgi-handler, 19  
comment-rule, 71  
continuation-id, 65  
continuation-URL, 71  
copy-ascii-port->port, 92  
copy-port->port-ascii, 92  
copy-port->port-binary, 92

default-rule, 71  
default-rules, 71  
display-low-level-sxml, 70  
dns-check-nameservers, 107  
dns-error, 104  
dns-error->string, 105  
dns-error?, 104  
dns-find-nameserver, 112  
dns-find-nameserver-list, 112  
dns-format-error, 105  
dns-format-error?, 105  
dns-get-information, 107  
dns-lookup, 107  
dns-lookup-ip, 106  
dns-lookup-mail-exchanger, 106  
dns-lookup-name, 106  
dns-lookup-nameserver, 106  
dns-message-cache?, 108  
dns-message-protocol, 108  
dns-message-query, 108  
dns-message-reply, 108  
dns-message-tried-nameservers,  
108  
dns-message?, 108  
dns-name-error, 105  
dns-name-error?, 105  
dns-not-implemented, 105  
dns-not-implemented?, 105  
dns-refused, 105  
dns-refused?, 105  
dns-server-error, 105  
dns-server-error?, 105  
dns-server-failure, 105  
dns-server-failure?, 105  
domains-for-search, 112

escape-uri, 24  
eval-safely, 20

- extract, 86
- extract-bindings, 73
- extract-single-binding, 73
- extract/single, 86
- final-page, 85
- flags-authoritative?, 109
- flags-opcode, 109
- flags-query-type, 109
- flags-recursion-available?, 109
- flags-recursion-desired?, 109
- flags-response-code, 109
- flags-truncated?, 109
- flags-zero, 109
- flags?, 109
- form-query, 85
- ftp-append, 91
- ftp-cd, 91
- ftp-cdup, 91
- ftp-connect, 90
- ftp-delete, 91
- ftp-dir, 91
- ftp-error?, 92
- ftp-get, 91
- ftp-inetd, 87
- ftp-ls, 91
- ftp-mkdir, 92
- ftp-modification-time, 92
- ftp-put, 91
- ftp-pwd, 91
- ftp-quit, 92
- ftp-quot, 92
- ftp-rename, 90
- ftp-rmdir, 91
- ftp-size, 92
- ftp-type, 90
- ftpd, 87
- generate-input-field-name, 79
- get-bindings, 73
- get-content-length, 73
- get-continuations, 64
- get-loaded-surfleets, 60
- get-session, 63
- get-session-data, 65
- get-sessions, 63
- header-additional-count, 109
- header-answer-count, 109
- header-flags, 108
- header-id, 108
- header-nameserver-count, 109
- header-question-count, 108
- header?, 108
- home-dir-handler, 18
- host-fqdn, 107
- http-url->string, 28
- http-url-fragment-identifier, 28
- http-url-path, 28
- http-url-search, 28
- http-url-server, 28
- http-url?, 28
- httpd, 9
- if-outdated, 83
- inform, 85
- input-field-attributes , 79
- input-field-binding, 74
- input-field-html-tree, 79
- input-field-html-tree-maker, 79
- input-field-multi?, 79
- input-field-name, 79
- input-field-transformer, 79
- input-field-type, 79
- input-field-value, 74
- instance-session-id, 63
- ip-string->address32, 113
- ip-string?, 113
- lifetime, 63
- loser, 20
- make-address, 81
- make-annotated-address, 81
- make-annotated-callback, 82
- make-annotated-checkbox, 76
- make-annotated-radio-group, 76
- make-annotated-select-option, 77

- make-boolean, 85
- make-callback, 81
- make-checkbox, 76
- make-error-response, 12
- make-file-directory-options, 18
- make-ftp-options, 88
- make-hidden-input-field, 75
- make-host-name-handler, 15
- make-http-url, 28
- make-httpd-options, 11
- make-image-button, 75
- make-input-field, 78
- make-multi-input-field, 78
- make-number, 85
- make-number-field, 74
- make-outdater, 83
- make-password, 85
- make-password-field, 74
- make-path-predicate-handler, 15
- make-path-prefix-handler, 15
- make-predicate-handler, 15
- make-query-message header, 108
- make-radio, 85
- make-radio-group, 76
- make-reader-writer-body, 14
- make-redirect-response, 12
- make-reset-button, 75
- make-response, 12
- make-select, 77
- make-server, 27
- make-simple-query-message, 108
- make-simple-select-option, 77
- make-submit-button, 75
- make-surplet-options, 58
- make-surplet-response, 62
- make-text, 85
- make-text-field, 74
- make-textarea, 74
- make-writer-body, 14
- make-yes-no, 85
- maybe-dns-lookup-ip, 106
- maybe-dns-lookup-name, 106
- message-additionals, 108
- message-answers, 108
- message-class, 109
- message-class-name, 109
- message-class-number, 109
- message-class?, 109
- message-header, 108
- message-nameservers, 108
- message-questions, 108
- message-source, 108
- message-type, 110
- message-type-name, 110
- message-type-number, 110
- message-type?, 110
- message?, 108
- my-continuation-id, 65
- my-ids, 65
- my-session-id, 65
- name->status-code, 13
- nbsp, 69
- nbsp-rule, 71
- netrc-entry-account, 94
- netrc-entry-login, 94
- netrc-entry-machine, 94
- netrc-entry-password, 94
- netrc-entry?, 94
- netrc-machine-entry, 93
- netrc-macro-definitions, 94
- network-protocol, 107
- network-protocol?, 107
- no-nameservers, 105
- no-nameservers?, 105
- not-a-hostname, 105
- not-a-hostname?, 105
- not-a-ip, 105
- not-a-ip?, 105
- null-request-handler, 15
- options-cache-surfleets?, 59
- options-make-session-timeout-text, 59
- options-session-lifetime, 59
- options-surplet-path, 59
- parse-error, 105

parse-error?, 105	resource-record-data, 110
parse-html-form-query, 21	resource-record-data-a-ip, 111
parse-http-url, 28	resource-record-data-a?, 111
parse-http-url-string, 29	resource-record-data-cname-name,
parse-server, 27	111
parse-uri, 23	resource-record-data-cname?, 111
plain-html, 69	resource-record-data-mx-exchanger,
plain-html-rule, 71	111
pop3-connect, 102	resource-record-data-mx-preference,
pop3-delete, 103	111
pop3-error?, 103	resource-record-data-mx?, 111
pop3-last, 103	resource-record-data-ns-name, 111
pop3-quit, 103	resource-record-data-ns?, 111
pop3-reset, 103	resource-record-data-ptr-name,
pop3-retrieve-headers, 103	111
pop3-retrieve-message, 103	resource-record-data-ptr?, 111
pop3-stat, 103	resource-record-data-soa-expire,
pretty-print-dns-message, 108	111
	resource-record-data-soa-minimum,
queries, 85	111
question-class, 109	resource-record-data-soa-mname,
question-name, 109	111
question-type, 109	resource-record-data-soa-refresh,
question?, 109	111
	resource-record-data-soa-retry,
raw-input-field-value, 74	111
read-rfc822-field, 95	resource-record-data-soa-rname,
read-rfc822-field-with-line-breaks,	111
95	resource-record-data-soa-serial,
read-rfc822-headers, 96	111
read-rfc822-headers-with-line-breaks,	resource-record-data-soa?, 111
96	resource-record-name, 110
request-handler, 14	resource-record-ttl, 110
request-headers, 12	resource-record-type, 110
request-method, 11	resource-record?, 110
request-socket, 12	resume-url-continuation-id, 72
request-uri, 11	resume-url-ids, 72
request-url, 11	resume-url-session-id, 72
request-version, 11	resume-url?, 72
request?, 11	returned-via, 80
resolv.conf, 112	returned-via?, 80
resolv.conf-parse-error, 112	rfc822-time->string, 96
resolv.conf-parse-error?, 112	rfc867-daytime/tcp, 97
resource-record-class, 110	rfc867-daytime/udp, 97

- rfc868-time/tcp, 98
- rfc868-time/udp, 98
- rooted-file-handler, 18
- rooted-file-or-directory-handler, 18
- select-option?, 77
- send, 66
- send-error, 66
- send-html, 67
- send-html/finish, 67
- send-html/suspend, 67
- send/finish, 66
- send/suspend, 66
- server->string, 27
- server-host, 27
- server-password, 27
- server-port, 27
- server-user, 27
- server?, 27
- session, 62
- session data, 65
- session-alive?, 63
- session-continuation-counter, 64
- session-continuation-table, 64
- session-continuation-table-lock, 64
- session-lifetime, 64
- session-session-id, 63
- session-surflet-name, 63
- seval-handler, 19
- show-outdated, 84
- simple-surflet-api, 84
- simplify-uri-path, 25
- single-query, 84
- smtp-connect, 100
- smtp-data, 101
- smtp-error?, 99
- smtp-expand, 100
- smtp-expn, 101
- smtp-get-help, 100
- smtp-helo, 101
- smtp-help, 101
- smtp-mail, 101
- smtp-noop, 101
- smtp-quit, 101
- smtp-rcpt, 101
- smtp-recipients-rejected-error?, 99
- smtp-rset, 101
- smtp-saml, 101
- smtp-send, 101
- smtp-send-mail, 99
- smtp-soml, 101
- smtp-transactions, 100
- smtp-transactions/no-close, 100
- smtp-turn, 101
- smtp-verify, 100
- smtp-vrfy, 101
- socket-address->fqdn, 106
- split-uri, 25
- status-code, 13
- status-code-message, 13
- status-code-number, 13
- surflet-file-name, 65
- surflet-form, 69
- surflet-form-rule, 71
- surflet-handler, 57
- surflet-handler/options, 58
- surflet-handler/primitives, 66
- surflet-handler/requests, 61
- surflet-handler/responses, 62
- surflet-handler/resume-url, 72
- surflet-handler/session-data, 66
- surflet-request-headers, 61
- surflet-request-input-port, 61
- surflet-request-method, 61
- surflet-request-request, 61
- surflet-request-socket, 61
- surflet-request-uri, 61
- surflet-request-url, 61
- surflet-request-version, 61
- surflet-request?, 61
- surflet-requests, 61
- surflet-response, 62
- surflet-response-content-type, 62
- surflet-response-data, 62
- surflet-response-headers, 62

- surflet-response-status, 62
- surflet-response?, 62
- surflet-sxml->low-level-sxml, 71
- surflet-sxml-rules, 71
- surfleets, 60
- surfleets/addresses, 81
- surfleets/bindings, 73
- surfleets/continuations, 64
- surfleets/ids, 65
- surfleets/input-field-value, 74
- surfleets/my-input-fields, 78
- surfleets/returned-via, 80
- surfleets/sessions, 62
- surfleets/surflet-input-fields, 74
- surfleets/surflet-sxml, 71
- surfleets/sxml, 68, 69, 71
- sxml->low-level-sxml, 69
- sxml->string, 70
- sxml-attribute-attributes, 68
- sxml-attribute?, 68
- SXML-rule, 70
- system-fqdn, 107
- text-rule, 71
- tilde-home-dir-handler, 18
- unescape-uri, 24
- unexpected-eof-from-server, 105
- unexpected-eof-from-server?, 105
- unload-surflet, 60
- uri-escaped-chars, 24
- uri-path->uri, 25
- url, 68
- url-rule, 71
- valid-surflet-response-data?, 62
- with-anonymous-home, 87
- with-back-icon-url, 17
- with-banner, 87
- with-blank-icon-url, 17
- with-cache-surfleets?, 58
- with-dns-lookup?, 88
- with-file-name->content-encoding, 17
- with-file-name->content-type, 17
- with-file-name->icon-url, 17
- with-fqdn, 10
- with-log-file, 10
- with-log-port, 87
- with-make-session-timeout-text, 58
- with-port, 10, 87
- with-reported-port, 10
- with-request-handler, 10
- with-resolve-ips?, 11
- with-root-directory, 10
- with-server-admin, 10
- with-session-lifetime, 58
- with-simultaneous-requests, 10
- with-surflet-path, 58
- with-syslog?, 11
- with-unknown-icon-url, 18