Guile-WWW Modules Reference

Table of Contents

The (www *) Modules

Guile-WWW is a set of Guile Scheme modules providing support for navigating HTTP connections, parsing URLs, handling CGI operations, and fetching WWW resources. This document corresponds to Guile-WWW 2.30.

1 (www http)

The `(www http)` module includes procedures for high-level HTTP operation, low-level HTTP message object access, and common messages.

1.1 High-Level HTTP Operation

`http:connect` *proto addrfam address* [*address-rest...*] [Procedure]
Return a TCP stream socket connected to the location specified by protocol *proto*, *addrfam* and *address*. *proto* is `PF_INET` or `PF_UNIX`, and the other args take corresponding forms:

`PF_INET` `(AF_INET` *ipaddr portno*`)`, where *ipaddr* is an integer. Use `(car (hostent:addr-list (gethost` *host*`)))` to compute the ipaddr of *host* (a string).

`PF_UNIX` `(AF_UNIX` *filename*`)`, made, for example, by `(list AF_UNIX "/tmp/foo-control")`.

Note that `PF_foo` and `AF_foo` are names of variables that have constant values, not symbols.

`http:open` *host* [*port*] [Procedure]
Return an HTTP connection (a socket) to *host* (a string) on TCP port *port* (default 80 if unspecified).

`http:request` *method url* [*headers* [*body*]] [Procedure]
Submit an HTTP request using *method* and *url*, wait for a response, and return the response as an HTTP message object.

method is the name of some HTTP method, e.g. `"GET"` or `"POST"`. *url* is a url object returned by `url:parse`. Optional args *headers* and *body* are lists of strings that comprise the lines of an HTTP message. The strings should not end with '`CR`' or '`LF`' or '`CRLF`'; `http:request` handles that. Also, the Content-Length header and Host header are calculated automatically and should not be supplied. Here are two examples:

```
(http:request "get" parsed-url
  (list "User-Agent: Anonymous/0.1"
        "Content-Type: text/plain"))

(http:request "post" parsed-url
  (list "User-Agent: Fred/0.1"
        "Content-Type: application/x-www-form-urlencoded")
  (list (string-append "search=Gosper"
                       "&case=no"
                       "&max_hits=50")))
```

As a special case (demonstrated in the second example above), when Content-Type is `application/x-www-form-urlencoded` and there is only one line in the body, the final '`CRLF`' is omitted and the Content-Length is adjusted accordingly.

1.2 Low-Level HTTP Message Object Access

`http:message-version` *msg* [Procedure]
Return the HTTP version in use in HTTP message *msg*.

`http:message-status-code` *msg* [Procedure]
Return the status code returned in HTTP message *msg*.

`http:message-status-text` *msg* [Procedure]
Return the text of the status line from HTTP message *msg*.

`http:message-status-ok?` *msg* [Procedure]
Return `#t` iff status code of *msg* indicates a successful request.

`http:status-ok?` *status* [Procedure]
Return `#t` iff *status* (a string) begins with `"2"`.

`http:message-body` *msg* [Procedure]
Return the body of the HTTP message *msg*.

An HTTP message header is represented by a pair. The CAR is a symbol representing the header name, and the CDR is a string containing the header text. E.g.:

```
'((date . "Thu, 29 May 1997 23:48:27 GMT")
  (server . "NCSA/1.5.1")
  (last-modified . "Tue, 06 May 1997 18:32:03 GMT")
  (content-type . "text/html")
  (content-length . "8097"))
```

Note: these symbols are all lowercase, although the original headers may be mixed-case. Clients using this library should keep this in mind, since Guile symbols are case-sensitive.

`http:message-headers` *msg* [Procedure]
Return a list of the headers from HTTP message *msg*.

`http:message-header` *header msg* [Procedure]
Return the header field named *header* from HTTP message *msg*, or `#f` if no such header is present in the message.

1.3 Common Messages

`http:head` *url* [Procedure]
Submit an http request using the `HEAD` method on the *url*. The `Host` header is automatically included.

`http:get` *url* [Procedure]
Submit an http request using the `GET` method on the *url*. The `Host` header is automatically included.

`http:post-form` *url extra-headers fields* [Procedure]

Submit an http request using the `POST` method on the *url*. *extra-headers* is a list of extra headers, each a string of form "*name*: *value* ...".

The "Content-Type" and "Host" headers are sent automatically and do not need to be specified. *fields* is a list of elements of the form `(fkey . fvalue)`, where *fkey* is a symbol and *fvalue* is normally a string.

fvalue can also be a list of file-upload specifications, each of which has the form `(source name mime-type transfer-encoding)`. *source* can be a string or a thunk that returns a string.

The rest of the elements are strings or symbols: *name* is the filename (only the non-directory part is used); *mime-type* is a type/subtype pair such as "image/jpeg", or `#f` to mean "text/plain". *transfer-encoding* is one of the tokens specified by RFC 1521, or `#f` to mean "binary". File-upload spec elements with invalid types result in a "bad upload spec" error prior to the http request.

Note that *source* is used directly without further processing; it is the caller's responsibility to ensure that the MIME type and transfer encoding specified describe *source* accurately.

2 (www url)

The `(www url)` module provides procedures for high-level url object conversion, low-level url object construction and access, and character decoding/encoding.

2.1 High-Level URL Object Conversion

`url:parse` *string* [Procedure]
Parse *string* and return a url object, with one of the following "schemes": HTTP, FTP, mailto, unknown.

`url:unparse` *url* [Procedure]
Return the *url* object formatted as a string. Note: The username portion is not included!

2.2 Low-Level URL Object Construction

`url:make` *scheme* [*args...*] [Procedure]
Construct a url object with specific *scheme* and other *args*. The number and meaning of *args* depends on the *scheme*.

`url:make-http` *host port path* [Procedure]
Construct a HTTP-specific url object with *host*, *port* and *path* portions.

`url:make-ftp` *user host port path* [Procedure]
Construct a FTP-specific url object with *user*, *host*, *port* and *path* portions.

`url:make-mailto` *address* [Procedure]
Construct a mailto-specific url object with an *address* portion.

2.3 Low-Level URL Object Access

`url:scheme` *url* [Procedure]
Extract and return the "scheme" portion of a *url* object. `url:scheme` is an unfortunate term, but it is the technical name for that portion of the URL according to RFC 1738. Sigh.

`url:address` *url* [Procedure]
Extract and return the "address" portion of the *url* object.

`url:unknown` *url* [Procedure]
Extract and return the "unknown" portion of the *url* object.

`url:user` *url* [Procedure]
Extract and return the "user" portion of the *url* object.

`url:host` *url* [Procedure]
Extract and return the "host" portion of the *url* object.

`url:port` *url* [Procedure]
Extract and return the "port" portion of the *url* object.

`url:path` *url* [Procedure]
Extract and return the "path" portion of the *url* object.

2.4 Character Decoding/Encoding

`url:decode` *str* [Procedure]
Re-export `url-coding:decode`. See Chapter 5 [url-coding], page 11.

`url:encode` *str reserved-chars* [Procedure]
Re-export `url-coding:encode`. See Chapter 5 [url-coding], page 11.

3 (www cgi)

The `(www cgi)` module provides procedures to support painlessly writing Common Gateway Interface scripts to process interactive forms. These scripts typically follow the following steps: initialization and discovery, data transfer in, data transfer out.

3.1 Initialization and Discovery

`cgi:init` [*opts...*] [Procedure]

(Re-)initialize internal data structures. This must be called before calling any other '`cgi:foo`' procedure. For FastCGI, call this "inside the loop" (that is, for each CGI invocation).

opts are zero or more symbols that configure the module.

`uploads-lazy`

This controls how uploaded files, as per `cgi:uploads` and `cgi:upload`, are represented.

Unrecognized options are ignored.

`cgi:form-data?` [Procedure]

Return `#t` iff there is form data available.

`cgi:names` [Procedure]

Return a list of variable names in the form. The order of the list is the same as that found in the form for the first occurance of each variable and each variable appears at most once. For example, if the form has variables ordered `a b a c d b e`, then the returned list would have order `a b c d e`.

`cgi:cookie-names` [Procedure]

Return a list of cookie names.

3.2 Data Transfer In

`cgi:getenv` *key* [Procedure]

Return the value of the environment variable associated with *key*, a symbol. Unless otherwise specified below, the return value is a (possibly massaged, possibly empty) string. The following keys are recognized:

- server-software-type
- server-software-version
- server-hostname
- gateway-interface
- server-protocol-name
- server-protocol-version
- server-port (integer)
- request-method
- path-info

- path-translated
- script-name
- query-string
- remote-host
- remote-addr
- authentication-type
- remote-user
- remote-ident
- content-type
- content-length (integer, possibly 0)
- http-accept-types (list, possibly empty, of strings)
- http-user-agent
- http-cookie

Keys not listed above result in an "unrecognized key" error.

`cgi:values` *name* [Procedure]
Fetch any values associated with *name* found in the form data. Return a list, even if it contains only one element. A value is either a string, or `#f`. When there are multiple values, the order is the same as that found in the form.

`cgi:value` *name* [Procedure]
Fetch only the CAR from `(cgi:values` *name*`)`. Convenient for when you are certain that *name* is associated with only one value.

`cgi:uploads` *name* [Procedure]
Return a list of file contents associated with *name*, or `#f` if no files are available.

Uploaded files are parsed by `parse-form` (see Chapter 9 [form-2-form], page 17). If the `uploads-lazy` option is specified to `cgi:init`, then the file contents are those directly returned by `form-2-form`. If unspecified, the file contents are strings with the object property `#:guile-www-cgi` whose value is an alist with the following keys:

`#:name` identical to *name* (sanity check)

`#:filename`
original/suggested filename for this bunch of bits

`#:mime-type`
something like "image/jpeg"

`#:raw-mime-headers`
the MIME headers before parsing

Note that the string's object property and the keys are all keywords. The associated values are strings.

Unless `uploads-lazy` is specified (to `cgi:init`), `cgi:uploads` can only be called once per particular *name*. Subsequent calls return `#f`. Caller had better hang onto the information, lest the garbage man whisk it away for good. This is done to minimize the amount of time the file is resident in memory.

`cgi:upload` *name* [Procedure]
Fetch the first file associated with form var *name*. Can only be called once per *name*, so the caller had better be sure that there is only one file associated with *name*. Use `cgi:uploads` if you are unsure.

`cgi:cookies` *name* [Procedure]
Fetch any cookie values associated with *name*. Return a list of values in the order they were found in the HTTP header, which should be the order of most specific to least specific path associated with the cookie. If no cookies are associated with *name*, return `#f`.

`cgi:cookie` *name* [Procedure]
Fetch the first cookie value associated with *name*.

3.3 Uncollated Form Data

With `cgi:values`, when a name occurs more than once, its associated values are collated, thus losing information about the relative order of different and intermingled names. For this, you can use `cgi:nv-pairs` to access the uncollated (albeit ordered) form data.

`cgi:nv-pairs` [Procedure]
Fetch the list of `(name . value)`, in the same order as found in the form data. A name may appear more than once. A value is either a string, or `#f`.

4 (www main)

The `(www main)` module provides a generic interface useful for retriving data named by any URL. The URL scheme `http` is pre-registered.

`www:set-protocol-handler!` *proto handler* [Procedure]
Associate for scheme *proto* the procedure *handler*. *proto* is a symbol, while *handler* is a procedure that takes three strings: the host, port and path portions, respectively of a url object. Its return value is the return value of `www:get` (for *proto*), and need not be a string.

`www:get` *url-string* [Procedure]
Parse *url-string* into portions. For HTTP, open a connection, retrieve and return the specified document. Otherwise, consult the handler procedure registered for the particular scheme and apply it to the host, port and path portions of *url-string*. If no such handler exists, signal "unknown URL scheme" error.

There is also the convenience proc `www:http-head-get`.

`www:http-head-get` *url-string* [*alist?*] [Procedure]
Parse *url-string* into portions; issue an "HTTP HEAD" request. Signal error if the scheme for *url-string* is not `http`. Optional second arg `alist?` non-`#f` means return only the alist portion of the HTTP response object.

5 (www url-coding)

The `(www url-coding)` module provides two procedures for decoding and encoding URL strings for safe transmission according to RFC 1738.

`url-coding:decode` *str* [Procedure]
: Return a new string made from url-decoding *str*. Specifically, turn `+` into space, and hex-encoded `%XX` strings into their eight-bit characters.

`url-coding:encode` *str reserved-chars* [Procedure]
: Return a new string made from url-encoding *str*, unconditionally transforming those in *reserved-chars*, a list of characters, in addition to those in the standard (internal) set.

6 (www utcsec)

The `(www utcsec)` module provides procedures to work with the *utc-seconds* of an object, that is, the number of seconds after epoch, in the GMT time zone (also known as UTC).

`format-utcsec` *port format utc-seconds* [Procedure]
Write to output port *port* the *utc-seconds* formatted according to *format* (a string). If *port* is `#f`, return the output string, instead. This uses `strftime`, q.v.

`rfc1123-date<-` *port utc-seconds* [Procedure]
Write to output port *port* the *utc-seconds* formatted according to RFC1123. If *port* is `#f`, return the output string, instead.

For example:

```
(rfc1123-date<- #f 1167791441)
⇒ "Wed, 03 Jan 2007 02:30:41 GMT"
```

`<-rfc1123-date` *s* [Procedure]
Parse the RFC1123-compliant date string *s*, and return the utc-seconds it represents.

For example:

```
(<-rfc1123-date "Wed, 03 Jan 2007 02:30:41 GMT")
⇒ 1167791441
```

`<-mtime` *filespec* [Procedure]
Return the utc-seconds of the modification time of *filespec*. *filespec* can be a filename (string), a port opened on a `stat`able file, or the object resulting from a `stat` on one of these.

For example:

```
(= (<-mtime "COPYING")
   (<-mtime (open-input-file "COPYING"))
   (<-mtime (stat "COPYING")))
⇒ #t
```

`<-ctime` *filespec* [Procedure]
Return the utc-seconds of the creation time of *filespec*. *filespec* can be a filename (string), a port opened on a `stat`able file, or the object resulting from a `stat` on one of these.

`rfc1123-now` [Procedure]
The "current time" formatted according to RFC1123.

7 (www server-utils big-dishing-loop)

The `(www server-utils big-dishing-loop)` module provides procedures that facilitate generation of a customized listener/dispatch proc.

`named-socket` *family name* [`keyword value...`] [Procedure]

Keywords: `socket-setup`

Return a new socket in protocol *family* with address *name*.

First, evaluate `(socket` *`family`* `SOCK_STREAM 0)` to create a new socket *sock*. Next, handle `#:socket-setup`, with value *setup*, like so:

`#f` Do nothing. This is the default.

procedure Call *procedure* on *sock*.

`((`*`opt`* `.` *`val`*`) ...)`
For each pair in this alist, call `setsockopt` on *sock* with the pair's *opt* and *val*.

Lastly, `bind` *sock* to *name*, which should be in a form that is appopriate for *family*. Two common cases are:

`PF_INET` `(AF_INET` *`ipaddr portno`*`)`, made, for example, by `(list AF_INET INADDR_ANY 4242)`.

`PF_UNIX` `(AF_UNIX` *`filename`*`)`, made, for example, by `(list AF_UNIX "/tmp/foo-control")`.

Note that `PF_foo`, `AF_foo`, and `INADDR_foo` are names of variables that have constant values, not symbols.

`echo-upath` *M upath* [*extra-args...*] [Procedure]
Use mouthpiece *M* (see Chapter 13 [answer], page 28) to compose and send a `"text/plain"` response which has the given *upath* (a string) and any *extra-args* as its content. Shut down the socket for both transmission and reception, then return `#t`.

This proc can be used to ensure basic network connectivity (i.e., aliveness testing).

`make-big-dishing-loop` [`keyword value...`] [Procedure]
Return a proc *dish* that loops serving http requests from a socket. *dish* takes one arg *ear*, which may be a pre-configured socket, a TCP port number, or a list of the form: `(`*`family address ...`*`)`. When *ear* is a TCP port number, it is taken to be the list `(PF_INET AF_INET INADDR_ANY` *`ear`*`)`.

In the latter two cases, the socket is realized by calling `named-socket` with parameters *family* and *name* taken from the CAR and CDR, respectively, of the list, with the `#:socket-setup` paramater (see below) passed along unchanged.

dish behavior is controlled by the keyword arguments given to `make-big-dishing-loop`. The following table is presented roughly in order of the steps involved in processing a request, with default values shown next to the keyword.

`#:socket-setup #f`
: This may be a proc that takes a socket, or a list of opt/val pairs which are passed to `setsockopt`. Socket setup is done for newly created sockets (when *dish* is passed a TCP port number), prior to the `bind` call.

`#:queue-length 0`
: The number of clients to queue, as set by the `listen` system call. Setting the queue length is done for both new and pre-configured sockets.

`#:concurrency #:new-process`
: The type of concurrency (or none if the value is not recognized). Here are the recognized values:

  `#:new-process`
  `#:new-process/nowait`
  : Fork a new process for each request. The latter does not wait for the child process to terminate before continuing the listen loop.

  `#f`
  : Handle everything in the current in process (no concurrency). Unrecognized values are treated the same as `#f`.

`#:bad-request-handler #f`
: If the first line of an HTTP message is not in the proper form, this specifies a proc that takes a mouthpiece *m*. Its return value should be the opposite boston value of the `#:loop-break-bool` value, below. See Chapter 13 [answer], page 28.

`#:method-handlers ()`
: This alist describes how to handle the (valid) HTTP methods. Each element has the form `(`*`method . handler`*`)`. *method* is a symbol, such as `GET`; and *handler* is a procedure that handles the request for *method*.

  handler normally takes two arguments, the mouthpiece *m* and the *upath* (string), composes and sends a response, and returns non-`#f` to indicate that the big dishing loop should continue.

  The proc's argument list is configured by `#:need-headers`, `#:need-input-port` and `#:explicit-return`. Interpretation of the proc's return value is configured by `#:explicit-return` and `#:loop-break-bool`. See below.

`#:need-headers #f`
`#:need-input-port #f`
: If non-`#f`, these cause additional arguments to be supplied to the handler proc. If present, the headers arg precedes the input port arg. See Chapter 8 [parse-request], page 16. The input port is always positioned at the beginning of the HTTP message body.

  If `#:need-input-port` is `#f`, after the handler proc returns, the port is `shutdown` in both (r/w) directions. When operating concurrently, this is done on the child side of the split. See Section "Network Sockets and Communication" in *The Guile Reference Manual*.

`#:explicit-return #f`
: If non-`#f`, this arranges for a continuation to be passed (as the last argument) to the handler proc, and ignores that proc's normal return value in favor of one explicitly passed through the continuation. If the continuation is not used, the *effective return value* is computed as `(not #:loop-break-bool)`.

`#:loop-break-bool #f`
: Looping stops if the effective return value of the handler is `eq?` to this value.

`#:unknown-http-method-handler #f`
: If `#f`, silently ignore unknown HTTP methods, i.e., those not specified in `#:method-handlers`. The value may also be a procedure that takes three arguments: a mouthpiece *m*, the *method* (symbol) and the *upath* (string). Its return value should be the opposite boolean value of the `#:loop-break-bool` value, below. See Chapter 13 [answer], page 28.

`#:parent-finish close-port`
: When operating concurrently (`#:concurrency` non-`#f`), the "parent" applies this proc to the port after the split.

`#:log #f`
: This proc is called after the handler proc returns. Note that if *ear* is a unix-domain socket, the *client* parameter will be simply `"localhost"`. See Chapter 14 [log], page 32.

`#:status-box-size #f`
: This may be a non-negative integer, typically 0, 1 or 2. It is used by `#:log` (has no meaning if `#:log` is `#f`). See Chapter 14 [log], page 32.

`#:style #f`
: An object specifying the syntax of the first-line and headers. The default specifies a normal HTTP message (see Chapter 1 [http], page 2).

The combination of `#:need-headers`, `#:need-input-port` and `#:explicit-return` mean that the `#:GET-upath` proc can receive anywhere from two to five arguments. Here is a table of all the possible combinations (1 means non-`#f` and 0 means `#f`):

```
+----- #:explicit-return
| +--- #:need-input-port
| | +- #:need-headers
| | |
| | |  args to #:GET-upath proc
=====  ============================
0 0 0  M upath
0 0 1  M upath headers
0 1 0  M upath in-port
0 1 1  M upath headers in-port
1 0 0  M upath return
1 0 1  M upath headers return
1 1 0  M upath in-port return
1 1 1  M upath headers in-port return
```

8 (www server-utils parse-request)

The `(www server-utils parse-request)` module provides procedures to read the first line, the headers and the body, of an HTTP message on the input port.

`read-first-line` *port* [Procedure]
: Parse the first line of the HTTP message from input *port* and return a list of the method, URL path and HTTP version indicator, or `#f` if the line ends prematurely or is otherwise malformed. A successful parse consumes the trailing '`CRLF`' of the line as well. The method is a symbol with its constituent characters upcased, such as `GET`; the other elements are strings. If the first line is missing the HTTP version, `parse-first-line` returns the default `"HTTP/1.0"`.

`hqf<-upath` *upath* [Procedure]
: Parse *upath* and return three values representing its hierarchy, query and fragment components. If a component is missing, its value is `#f`.

```
(hqf<-upath "/aa/bb/cc?def=xyz&hmm#frag")
⇒ #<values "/aa/bb/cc" "def=xyz&hmm" "frag">

(hqf<-upath "/aa/bb/cc#fr?ag")
⇒ #<values "/aa/bb/cc" #f "fr?ag">
```

`alist<-query` *query-string* [Procedure]
: Parse urlencoded *query-string* and return an alist. For each element `(`*name* `.` *value*`)` of the alist, *name* is a string and *value* is either `#f` or a string.

`read-headers` *port* [Procedure]
: Parse the headers of the HTTP message from input *port* and return a list of key/value pairs, or `#f` if the message ends prematurely or is otherwise malformed. Both keys and values are strings. Values are trimmed of leading and trailing whitespace and may be empty. Values that span more than one line have their `"continuation whitespace"` reduced to a single space. A successful parse consumes the trailing '`CRLF`' of the header block as well.

Sometimes you are interested in the body of the message but not the headers. In this case, you can use `skip-headers` to quickly position the port.

`skip-headers` *port* [Procedure]
: Scan without parsing the headers of the HTTP message from input *port*, and return the empty list, or `#f` if the message ends prematurely. A successful scan consumes the trailing '`CRLF`' of the header block as well.

`read-body` *len port* [Procedure]
: Return a new string of *len* bytes with contents read from input *port*.

9 (www server-utils form-2-form)

The `(www server-utils form-2-form)` module provides a procedure to parse a string in '`multipart/form-data`' format.

`parse-form` *content-type-more raw-data* [Procedure]

Parse *raw-data* as raw form response data of enctype '`multipart/form-data`' and return an alist.

content-type-more is a string that should include the `boundary="..."` information. (This parameter name reflects the typical source of such a string, the Content-Type header value, after the '`multipart/form-data`'.)

Each element of the alist has the form `(`*`name . value`*`)`, where *name* is a string and *value* is either a string or four values (extractable by `call-with-values`):

filename A string, or `#f`.

type A string representing the MIME type of the uploaded file.

raw-headers
A string, including all eol CRLF chars. Incidentally, the *type* should be (redundantly) visible in one of the headers.

squeeze A procedure that takes one arg *abr* (standing for access byte range). If *abr* is `#f`, then internal references to the uploaded file's data are dropped. Otherwise, *abr* should be a procedure that takes three arguments: a string, a beginning index (integer, inclusive), and an ending index (integer, exclusive).

If there is no type information, *value* is a simple non-empty string, and no associated information (filename, raw-headers, squeeze) is kept.

`parse-form` ignores *degenerate uploads*, that is those parts of `raw-data` where the part header specifies no filename and the part content-length is zero or unspecified.

why `squeeze`?

The *squeeze* interface can help reduce data motion. Consider a common upload scenario: client uploads file(s) for local (server-side) storage.

```
classic  squeeze
   *        *       0. (current-input-port)
   *        *       1. Guile-WWW string (for parsing purposes)
   *                2. your substring (image/jpeg)
   *        *       3. filesystem
```

You can achieve the same effect as the "classic" approach by specifying `substring` (or something like it) as the access-byte-range proc, but **you don't have to**. You could, instead, call *squeeze* with a procedure that writes the byte range directly to the filesystem.

10 (www server-utils filesystem)

The `(www server-utils filesystem)` module provides procedures for cleaning filenames, checking filesystem access, and mapping from a URL path to a filename.

`cleanup-filename` *name* [Procedure]

Return a new filename made from cleaning up filename *name*. Cleaning up is a transform that collapses each of these, in order:

- '`//`'
- '`/./`'
- '`/foo/../`'

into a single slash ('`/`'), everywhere in *name*, plus some fixups. The transform normally preserves the trailing slash (if any) in *name*, and does not change any leading '`..`' components if *name* is relative, i.e., does not begin with slash. Due to proper '`/foo/../`' cancellation for relative *name*, however, the result may be the empty string. (Here, *proper* means that *foo* is not '`..`', but a normal filename component.)

Following is a fairly comprehensive list of the `cleanup-filename` edge cases, paired by *name* and result. The numbers represent string lengths.

```
0                       ;; empty string
0                       ;; result is empty string

1  /
1  /

2  ok
2  ok

3  ok/
3  ok/

3  /ok
3  /ok

4  /ok/
4  /ok/

1  .                    ;; relative name
0                       ;; result is empty string

2  ./                   ;; likewise
0                       ;; note, end-slash not preserved

2  /.
1  /
```

```
3  /./
1  /

2  ..                   ;; relative, with leading double-dot
2  ..                   ;; unchanged

3  ../                  ;; likewise
3  ../

3  /..                  ;; absolute
1  /                    ;; can't go higher than root

4  /../
1  /

4  ./..                 ;; next 8 are like the previous 4;
2  ..                   ;;  they show that  .  makes no difference

5  ./../
3  ../

5  /./..
1  /

6  /./../
1  /

4  ../.
2  ..

5  .././
3  ../

5  /../.
1  /

6  /.././
1  /

5  ../..               ;; relative
5  ../..               ;; leading  ..  sequences unchanged

6  ../../
6  ../../

6  /../..              ;; absolute
1  /                    ;; can't go higher than root
```

```
 7  /../../
 1  /

 4  z/..                ;; relative
 0                      ;; only dir cancelled ⇒ empty string

 5  z/../               ;; likewise
 0

 5  /z/..               ;; absolute
 1  /

 6  /z/../
 1  /

 6  z/../o             ;; next 4 like previous 4, with trailing component
 1  o

 7  z/../o/
 2  o/

 7  /z/../o
 2  /o

 8  /z/../o/
 3  /o/

 8  z/./../o           ;; next 4 like previous 4;
 1  o                   ;;  they show that  .  makes no difference

 9  z/./../o/
 2  o/

 9  /z/./../o
 2  /o

10  /z/./../o/
 3  /o/

 9  z/../../o          ;; relative, more double-dot than parents
 4  ../o                ;; leftover double-dot preserved

10  z/../../o/
 5  ../o/

10  /z/../../o         ;; absolute, more double-dot than parents
```

```
 2  /o                    ;; all cancelled

11  /z/../../o/
 3  /o/

43  ../../abc/./bye0/./../def/bye1/bye2/../..        ;; bye bye-bye
14  ../../abc/def/

44  ../../abc/./bye0/./../def/bye1/bye2/../../
14  ../../abc/def/

44  /../../abc/./bye0/./../def/bye1/bye2/../..
 9  /abc/def/

45  /../../abc/./bye0/./../def/bye1/bye2/../../
 9  /abc/def/
```

`access-forbidden?-proc` *docroot forbid-rx* [Procedure]

Create and return a filesystem-access procedure based on *docroot* and *forbid-rx*. The returned procedure *p* takes a *filename* and returns `#t` if access to that file should be denied for any of the following reasons:

- *filename* does not begin with *docroot*
- *filename* matches regular expression *forbid-rx*

If *forbid-rx* is `#f`, the regular expression check is skipped. *p* returns `#f` if access should be granted.

`upath->filename-proc` *docroot* [*dir-indexes*] [Procedure]

Create and return a url-path-to-filename mapping procedure based on *docroot*. The returned procedure *p* takes a (string) *upath* and returns a valid local filename path for the requested resource, or `#f` if that file cannot be found. Optional arg *dir-indexes* specifies an ordered list of filenames to try if the resolved filename path turns out to be a directory.

If no such files exist, return the directory name. As a special case, when *p* encounters a value of `#f` during iteration over *dir-indexes*, it returns `#f` immediately.

For example, presuming files '`/a/b/c.txt`' and '`/a/b/index.html`' both exist and are readable:

```
(define resolve (upath->filename-proc
                  "/a/b/"
                  '("index.shtml" "index.html")))

(resolve "/random") ⇒ #f
(resolve "/c.txt") ⇒ "/a/b/c.txt"
(resolve "/") ⇒ "/a/b/index.html"
```

Directory names are always returned with a trailing slash.

`filename->content-type` *filename* [*default*] [Procedure]

Return a valid Content-Type string which matches *filename* best. Matching is done by comparing the extension (part of *filename* after the last `"."` if available) against a table. If none match, return `"application/octet-stream"`. Optional arg *default* specifies another value to use instead of `"application/octet-stream"`.

If there are multiple MIME types associated with the extension, return the first one.

See Chapter 17 [mime-types], page 35, proc `put-mime-types!`, for more info.

11 (www server-utils cgi-prep)

Often the server cannot do everything by itself, and makes use of external programs invoked in a *common gateway interface* environment. These programs are also known as *CGI scripts*.

The `(www server-utils cgi-prep)` module provide a procedure to set up such an environment. Actually invoking the CGI script is not covered.

`cgi-environment-manager` *initial-bindings* [Procedure]

Return a closure encapsulating *initial-bindings*, a list of pairs `(name . value)`, where *name* is a symbol listed in the following table, and *value* is a string unless otherwise noted.

- `server-hostname`
- `gateway-interface`
- `server-port` (integer)
- `request-method`
- `path-info`
- `path-translated`
- `script-name`
- `query-string`
- `remote-host`
- `remote-addr`
- `authentication-type`
- `remote-user`
- `remote-ident`
- `content-type`
- `content-length` (integer, or `#f`)
- `http-user-agent`
- `http-cookie`
- `server-software`
- `server-protocol`
- `http-accept-types` (list of strings)

If *name* is not recognized, signal `"unrecognized key"` error. Encapsulation includes *`name=value`* formatting.

The closure accepts these commands:

`name value`
: Encapsulate an additional binding. *name* and *value* are as above.

`#:clear!`
: Drop the additional bindings. Note that initial bindings can never be dropped (you can always create a new closure).

`#:environ-list`
: Return a list of strings suitable for passing to `environ` or as the second argument to `execle`.

Any other command results in a `"bad command"` error.

example

Following is a simple example of how to use `cgi-environment-manager`. A more realistic example would include port and connection management, input validation, error handling, logging, etc. First, we set up the manager with more-or-less constant bindings.

```
(define M (cgi-environment-manager
            '((server-software . "FooServe/24")
              (server-protocol . "HTTP/1.0")
              (server-port . 80))))
```

Later, we add connection-specific bindings. We use `read-first-line` from the Chapter 8 [parse-request], page 16 module.

```
(define PORT ...)
(define UPATH (list-ref (read-first-line PORT) 1))
(define QMARK (string-index UPATH #\?))
(define CGI (substring UPATH 0 QMARK))

(M 'script-name CGI)
(M 'query-string (substring UPATH (1+ QMARK)))
```

Lastly, we spawn the child process, passing the constructed environment as the second arg to `execle`, and drop the connection-specific bindings afterwards.

```
(let ((pid (primitive-fork)))
  (if (zero? pid)
      (execle CGI (M #:environ-list) (list CGI)) ; child
      (waitpid pid)))                            ; parent

(M #:clear!)
```

Now we can re-use `M` for another connection.

12 (www server-utils cookies)

Cookies are bits of client-side state the server can maintain through designated HTTP response headers. At this time (2009), there are two specifications, RFC2109[1] and RFC2965[2], the latter obsoleting the former.

This chapter describes the `(www server-utils cookies)` module, which provides facilities for creating such headers, and parsing those sent by the client. Procedures that return trees are meant to be used with the `mouthpiece` command `#:add-header` (see Chapter 13 [answer], page 28).

`simple-parse-cookies` *string* [Procedure]

Parse *string* for cookie-like fragments using the simple regexp:

```
(,[ \t]*)*([^=]+)=([^,]+)
```

Return a list of elements `(name . value)`, where both *name* and *value* are strings. For example:

```
(simple-parse-cookies "abc=def; z=z, ans=\"42\", abc=xyz")
⇒ (("abc" . "def; z=z") ("ans" . "\"42\"") ("abc" . "xyz"))
```

`rfc2109-set-cookie-string` *name value* [`keyword value`...] [Procedure]

Keywords: `path`, `domain`, `expires`, `secure`

Return a string suitable for inclusion into an HTTP response header as a cookie with *name* and *value*. Both args may be strings, symbols or keywords. Also, recognize and format appropriately the optional keyword parameters `#:path`, `#:domain`, `#:expires` (strings); and `#:secure` (boolean).

`rfc2965-set-cookie2-tree` *M* [*cookie-specs*...] [Procedure]

Compute a list suitable for inclusion in an HTTP response header, composed by formatting *cookie-specs*, each a list of the form `(name value a1 v1...)`. Each *name* may be a string, symbol or keyword. Each *value* may be a string or symbol. Each *a* must be a keyword, precisely one of:

```
#:Comment  #:CommentURL  #:Discard  #:Domain
#:Max-Age  #:Path  #:Port  #:Secure
```

The #:Version attribute is automatically included as the last one; it cannot be specified (or de-specified).

Possible values for *v* depend on *a*. If *a* is `#:Discard` or `#:Secure`, then there is no *v* (it must be omitted). If *a* is `#:Port`, then *v* must be either a number; a list of numbers, for instance `(8001 8002 8003)`; or omitted entirely. If *a* is `#:Max-Age`, then *v* must be a number. For all other *a*, *v* can be a string or symbol.

If *M* is `#f`, return a list. The CAR of the list is the keyword `#:Set-Cookie2`, and the CDR is a tree of strings. Otherwise *M* should be a `mouthpiece` (see Chapter 13 [answer], page 28) in which case it is applied with the `#:add-header` command to the list.

[1] RFC2109

[2] RFC2965

example

Here is an example that demonstates both RFC2109 and RFC2965 formatting. Notable differences: the keyword to specify the path is now capitalized; the representation of the cookie's value is now double-quoted.

```
;; RFC2109
(rfc2109-set-cookie-string 'war 'lose #:path "/ignorance/suffering")
⇒ "Set-Cookie: war=lose; path=/ignorance/suffering"

;; RFC2965
(use-modules ((www server-utils answer) #:select (walk-tree)))

(define TREE (rfc2965-set-cookie2-tree
              '(war lose #:Path "/ignorance/suffering" #:Discard)))

(car TREE)
⇒ #:Set-Cookie2

(walk-tree display (cdr TREE))
⊣ war="lose";Path="/ignorance/suffering";Discard;Version=1
```

To generate a cookie spec from the `Cookie` http response header sent by a client, you can use `rfc2965-parse-cookie-header-value`.

`rfc2965-parse-cookie-header-value` *s* [*flags...*] [Procedure]

Parse the `Cookie` HTTP response header string *s*. Return a list of the form `(`*`vers n`* `[`*`cookie-spec...`*`])`, where *vers* is the version number of the cookie specification, 0 (zero) for RFC2109 compliance and 1 (one) for RFC2965 compliance; and *n* is the number of cookie-specs the CDR of the form.

Each *cookie-spec* has the form: `(`*`name value a1 v1...`*`)`. *name*, *value* are strings. Each *a* is a keyword, one of `#:Path`, `#:Domain` or `#:Port`. Each *v* is a string, except for that associated with `#:Port`, which is can be either a single number or a list of numbers.

Optional *flags* configure the parsing and/or return value.

`#:keep-attribute-dollarsign-prefix`
: Prevent conversion of, for example, `#:$Port` to `#:Port`.

`#:strict-comma-separator`
: Disable support for older clients that use a semicolon to separate cookies instead of a comma. Normally, parsing copes (heuristically) with this by reparsing an unrecognized attribute as the beginning of a new cookie. With this flag, an unrecognized attribute signals an error.

`#:canonicalize-NAME-as-keyword`
: Convert the *name* in each cookie-spec into a keyword whose first character and characters following a hyphen are upcased. For example, `"session-id-no"` would become `#:Session-Id-No`.

Parsing may signal an error and display an error message in the form: "*situation* while *context*", where *situation* is one of "unexpected end", "missing equal-sign", "bad

attribute", or "missing semicolon"; and *context* is one of: "reading string", "reading token", "reading pair", "reading one cookie" or "parsing". The error message also displays string *s* on a line by itself and on the next line a caret by itself indented to be at (or near) the site of the error.

RFC2965 also specifies some other small algorithms, some of which are codified as procedures available in this module.

`reach` *h* [Procedure]

Return the *reach* (a string) of host name *h*. Quoting from RFC2965 section 1 (Terminology):

The reach R of a host name H is defined as follows:
If
- H is the host domain name of a host; and,
- H has the form A.B; and
- A has no embedded (that is, interior) dots; and
- B has at least one embedded dot, or B is the string "local".

then the reach of H is .B.
Otherwise, the reach of H is H.

Note that comparison with "local" uses `string=?`, i.e., case-sensitively.

13 (www server-utils answer)

The `(www server-utils answer)` module provides a simple wrapper around the formatting/accounting requirements of a standard HTTP response. Additionally, the `#:rechunk-content` facility allows some degree of performance tuning; a server may be able to achieve better throughput with certain chunk sizes than with others.

The output from `mouthpiece` and `string<-headers` is formatted according to their optional *style* argument. By default, headers have the form:

```
NAME #\: #\space VALUE #\cr #\lf
```

Additionally, for `mouthpiece`, the first line, preceding all the headers, has the form:

```
HTTP/1.0 nnn msg
```

and a single `#\cr #\lf` pair separates the headers from the body. See Chapter 15 [modlisp], page 33, for another way to format this information.

`mouthpiece` *out-port* [*status-box* [*style*]] [Procedure]

Return a command-delegating closure capable of writing a properly formatted HTTP 1.0 response to *out-port*. Optional arg *status-box* is a list whose CAR is set to the numeric status code given to a `#:set-reply-status` command. If *status-box* has length of two or more, its CADR is set to the content-length on `#:send-reply`. A content-length value of `#f` means there have been no calls to `#:add-content`. The commands and their args are:

`#:reset-protocol!`
: Reset internal state, including reply status, headers and content. This is called automatically by `#:send-reply`.

`#:set-reply-status` *number message*
: Set the reply status. *message* is a short string.

`#:set-reply-status:success`
: This is equivalent to `#:set-reply-status 200 "OK"`.

`#:add-header` *name value*
: *name* may be `#f`, `#t`, a string, symbol or keyword. *value* is a string. If *name* is `#f` or `#t`, *value* is taken to be a pre-formatted string, `"A: B"` or `"A: B\r\n"`, respectively. If *name* is not a boolean, *value* may also be a tree of strings or a number.

`#:add-content [`*tree* `...]`
: *tree* may be a string, a nested list of strings, or a series of such. Subsequent calls to `#:add-content` append their trees to the collected content tree thus far.

`#:add-formatted` *format-string* `[`*args* `...]`
: *format-string* may be `#f` to mean `~S`, `#t` to mean `~A`, or a normal format string. It is used to format *args*, and the result passed to `#:add-content`.

`#:add-direct-writer` *len write*
: *len* is the number of bytes that procedure *write* will output to its arg, *out-port* (passed back), when called during `#:send-reply`. This is to allow sendfile(2) and related hackery.

`#:content-length`
: Return the total number of bytes in the content added thus far.

`#:rechunk-content` *chunk*
: *chunk* may be `#f`, in which case a list of the string lengths collected thus far is returned; `#t` which means to use the content length as the chunk size (effectively producing one chunk); or a number specifying the maximum size of a chunk. The return value is a list of the chunk sizes.

  It is an error to use `#:rechunk-content` with a non-`#f` *chunk* in the presence of a previous `#:add-direct-writer`.

`#:inhibit-content!` *bool*
: Non-`#f` *bool* arranges for `#:send-reply` (below) to compute content length and add the appropriate header, as usual, but no content is actually sent. This is useful, e.g., when answering a `HEAD` request. If *bool* is `#f`, `#:send-reply` acts normally (i.e., sends both headers and content).

`#:send-reply [close]`
: Send the properly formatted response to *out-port*, and reset all internal state (status reset, content discarded, etc). It is an error to invoke `#:send-reply` without having first set the reply status.

  Optional arg *close* means do a `shutdown` on *out-port* using *close* — directly, if an integer, or called with no arguments, if a thunk — as the shutdown `how` argument. (Note: If *out-port* is not a socket, this does nothing silently.) See ⟨undefined⟩ [Network Sockets and Communication], page ⟨undefined⟩.

  If *close* is specified, the closure forgets about *out-port* internally; it is an error to call other mouthpiece commands, subsequently.

example

Here is an example that uses most of the `mouthpiece` commands:

```
(use-modules (www server-utils filesystem) (scripts slurp))

(define SERVER-NAME "Guile-WWW-example-server")
(define SERVER-VERSION "1.0")
(define STATUS (list #f #f))
(define M (mouthpiece (open-output-file "fake") STATUS))

(define (transmit-file filename)
  (M #:set-reply-status:success)
  (M #:add-header #:Server (string-append SERVER-NAME " "
                                          SERVER-VERSION))
  (M #:add-header #:Connection "close")
  (M #:add-header #:Content-Type (filename->content-type
                                  filename "text/plain"))
  (M #:add-content (slurp filename))
  (simple-format #t "rechunked: ~A\n"
```

```
                (M #:rechunk-content (* 8 1024)))
  ;; We don't shutdown because this is a file port;
  ;; if it were a socket, we might specify 2 to
  ;; stop both reception and transmission.
  (M #:send-reply))

(transmit-file "COPYING")
⊣ rechunked: (8192 8192 1605)
STATUS
⇒ (200 17989)
```

For higher performance, you can preformat parts of the response, using `CRLF`, and some lower-level convenience procedures. If preformatting is not possible (or desirable), you can still declare a nested list of strings (aka *tree*) to have a *flat length*, i.e., the size in bytes a tree would occupy once flattened, thus enabling internal optimizations. (The flat length of a string is its `string-length`.)

`CRLF` [Constant String]
The string "\r\n".

`flat-length` *object* [Object Property]
Return the flat length of *object*, or `#f` if not yet computed.

`fs` *s* [*args...*] [Procedure]
Return a new string made by using format string *s* on *args*. As in `simple-format` (which this procedure uses), `~A` expands as with `display`, while `~S` expands as with `write`.

`walk-tree` *proc tree* [Procedure]
Call *proc* for each recursively-visited leaf in *tree*, excluding empty lists. It is an error for *tree* to contain improper lists.

`tree-flat-length!` *tree* [Procedure]
If *tree* is a string, return its `string-length`. If *tree* already has a `flat-length`, return that. Otherwise, recursively compute, set, and return the `flat-length` of *tree*.

`string<-tree` *tree* [Procedure]
Return a new string made from flattening *tree*. Set the `flat-length` (using `tree-flat-length!`) of *tree* by side effect.

`string<-headers` *alist* [*style*] [Procedure]
Return a string made from formatting name/value pairs in *alist*, according to the optional `style` argument. If unspecified or specified as `#f`, the default is to format headers like so:

```
NAME #\: #\space VALUE #\cr #\lf
```

Each name may be a string, symbol or keyword. Each value may be a string, number, symbol, or a tree.

`string<-header-components` *n v* [*n1 v1...*] [Procedure]

Return a string made from formatting header name *n* and value *v*. Additional headers can be specified as alternating name and value args. Each header is formatted like so: "*name*: *value*\r\n".

Each *n* may be a string, symbol or keyword. Each *v* may be a string, number, symbol, or a tree.

NOTE: This proc **will be removed** after 2011-12-31. Use `string<-headers` instead.

example

Here is `transmit-file` from the above example, slightly modified to use preformatted headers and `fs`:

```
(define CONSTANT-HEADERS
  (string<-headers
   `((#:Server     . ,(fs "~A ~A" SERVER-NAME SERVER-VERSION))
     (#:Connection . "close"))))

(define (transmit-file filename)
  (M #:set-reply-status:success)
  (M #:add-header #t CONSTANT-HEADERS)
  (M #:add-header #:Content-Type (filename->content-type
                                  filename "text/plain"))
  (M #:add-content (slurp filename))
  (display (fs "rechunked: ~A\n" (M #:rechunk-content (* 8 1024))))
  (M #:send-reply))
```

Note that `mouthpiece` accepts trees for both `#:add-header` and `#:add-content` commands. Thus, the following two fragments give the same result, although the latter is both more elegant and more efficient:

```
;; Doing things "manually".
(walk-tree (lambda (string)
             (M #:add-content string))
           tree)

;; Letting the mouthpiece handle things.
(M #:add-content tree)
```

14 (www server-utils log)

The `(www server-utils log)` module provides procedure generators for writing log information to an output port. Each generator is conventionally named `log-SOMETHING-proc`.

`log-http-response-proc` *port* [*gmtime?* [*stamp-format* [*method-pair?*]]] [Procedure]

Return a procedure that writes an HTTP response log entry to *port*. The procedure is called with args *client*, *method*, *upath* (strings or symbols) and *status* (either an atom or a list), and writes a one-line entry of the form:

```
CLIENT - - [YYYY-MM-DD:HH:MM:SS TZ] "METHOD UPATH" ST1 ST2...
```

where the '`YYYY..TZ`' are the year, month, day, hour, minute, second and timezone components, respectively, of the `localtime` representation of the current time; and '`STn`' are the space-separated elements of *status*.

Optional second arg *gmtime?* non-`#f` means use `gmtime` instead of `localtime`. Optional third arg *stamp-format* specifies a format string passed to `strftime` to use for the timestamp portion that appears between the square braces (default: `"%Y-%m-%d:%H:%M:%S %Z"`).

Optional fourth arg *method-pair?* non-`#f` means that *method* is expected to be a pair `(meth . vers)`, in which case the portion between the double quotes becomes "*meth upath vers*". This is to support excruciating conformity to Apache for the benefit of downstream programs that might fall over less than gracefully otherwise. Please enjoy the slack.

The buffering mode for *port* is set to line-buffered.

15 (www server-utils modlisp)

The `(www server-utils modlisp)` module provides support for the implementing the Lisp side of the Apache mod_lisp protocol, in the form of a header-grokking protocol object for the big dishing loop, and a style elements object for the mouthpiece. When these objects are specified, the headers are read from (written to) the Apache front end in the form:

```
name #\lf value #\lf
```

with a lone '`end\n`' to separate the headers from the body. Furthermore, on input, the headers must include `method`, `url` and `server-protocol`. On output, the status information (always output first) has the form:

```
"Status" #\lf nnn #\space msg #\lf
```

Note that this is in essense the same format as used for the headers, with *name* being '`Status`' and *value* being '*`nnn msg`*'.

`modlisp-hgrok` [Object]
: An object suitable for the value of `make-big-dishing-loop` keyword argument `#:style`. See Chapter 7 [big-dishing-loop], page 13.

`modlisp-ish` [Object]
: An object suitable as the optional `style` argument for both `string<-headers` and `mouthpiece`. See Chapter 13 [answer], page 28.

Although these are separate objects, you should probably use or not use them in conjunction, lest the front-end (Apache) server become confused.

16 (www data http-status)

The `(www data http-status)` module exports a single procedure:

`http-status-string` *number* [Procedure]
Return the string associated with HTTP status *number*.

example

Here is a simple example using this module:

```
(use-modules ((www data http-status)
              #:select (http-status-string)))

(define (h2 n)
  (format #f "<H2>~A ~A</H2>"
          n (http-status-string n)))

(h2 404) ⇒ "<H2>404 Not Found</H2>"
(h2 307) ⇒ "<H2>307 Temporary Redirect</H2>"
```

17 (www data mime-types)

The `(www data mime-types)` module maintains an internal hash table mapping filename extensions to one or more *mime-types*.

The exported procedures provide convenience abstractions over the underlying hash-table manipulation operations, including extension and mime-type validation, init from a file in a "standard" format (i.e., that of '`/etc/mime.types`' or '`~/.mime.types`'), and support for straightforward incremental init (aka *merging*). There are two predefined entries in the hash table:

```
text => text/plain
html => text/html
```

To support merging, the `put-FOO` procedures both take a symbol *resolve* as the first arg, which specifies how *conflicts* should be handled. This happens when the hash table already contains an entry for *extension* and *new-mime-type* differs from *old-mime-type*.

`error` Throw an error with key `mime-type-conflict`, displaying a message describing the *extension*, *old-mime-type* and *new-mime-type*.

`prefix` Make the mime-type of *extension* a list (unless already one), with *new-mime-type* at the beginning.

`suffix` Make the mime-type of *extension* a list (unless already one), with *new-mime-type* at the end.

`stomp` Use *new-mime-type* directly, discarding *old-mime-type*.

`quail` Discard *new-mime-type*, keeping *old-mime-type*.

For any other method, the operation throws an error, with key `invalid-resolve`.

Validation happens on all "put" operations. The extension must be a symbol, such as `txt`. The mime-type must be a symbol with exactly one '`/`' (slash) in its name, such as `text/plain`, or a proper list of such symbols. The mime-type may also be `#f`, which means to remove *extension* from the hash table.

If an entry does not validate, the operation throws an error, with key `invalid-extension` or `invalid-mime-type`.

`reset-mime-types!` *size* [Procedure]
Clear all entries from the mime-types hash table, and prepare it for *size* (approximately) entries. This procedure must be called before any others in this module.

`put-mime-types-from-file!` *resolve filename* [Procedure]
Open *filename* and parse its contents as "mime-types" format. This line-oriented file format is briefly described as follows:

- Blank lines and lines beginning with '`#`' are ignored.
- Lines of the format *mime-type* (only one symbol) are ignored.
- Otherwise, the line is expected to be in the format *`mime-type extension extension...`*, that is, at least one *extension* must be present. Each *extension* results in an entry in the hash table.

Put those those entries that specify an extension into the hash table, validating both extension and mime-type first. *resolve* specifies how to resolve extension conflicts.

`put-mime-types!` *resolve* [*extension1 mime-type1 ...*] [Procedure]
Put *extension1*/*mime-type1*... into the hash table, validating both extension and mime-type first. *resolve* specifies how to resolve extension conflicts.

If an extension is given but there is no mime-type (i.e., the list has an odd length), throw an error with key `missing-mime-type`.

`mime-types<-extension` *ext* [Procedure]
Return the mime-type(s) associated with *ext* (a symbol or string), or `#f` if none are found. Note that generally the value may be a single mime-type or a list of them.

`select-extensions` *sel* [Procedure]
Return a list of extensions in the hash table that match the *sel* criteria (a symbol). If *sel* is `#t`, return all the extensions; if `single`, only those who have a single mime-type associated; if `multiple`, only those who have more than one mime-type associated.

why `select-extensions`?

The last procedure is intended to ease non-generalizable merging, without providing too much exposure to implementation internals. Suppose you want to maintain a local policy of having only one mime-type associated per extension (to keep things simple). In that case, after populating the hash, you can fix up those entries, like so:

```
(reset-mime-types! 491)
(put-mime-types-from-file! 'prefix "/etc/mime.types")
(define AMBIGUOUS (select-extensions 'multiple))

(use-modules (ice-9 format))
(define (display-ext ext)
  (format #t "~7,@A  ~A~%" ext (mime-types<-extension ext)))

(for-each display-ext AMBIGUOUS)
    ent  (chemical/x-ncbi-asn1-ascii chemical/x-pdb)
    sdf  (application/vnd.stardivision.math chemical/x-mdl-sdfile)
     sh  (application/x-sh text/x-sh)
    csh  (application/x-csh text/x-csh)
    cpt  (application/mac-compactpro image/x-corelphotopaint)
    asn  (chemical/x-ncbi-asn1 chemical/x-ncbi-asn1-spec)
    wrl  (model/vrml x-world/x-vrml)
    tcl  (application/x-tcl text/x-tcl)
     ra  (audio/x-pn-realaudio audio/x-realaudio)
    spl  (application/futuresplash application/x-futuresplash)
    m3u  (audio/mpegurl audio/x-mpegurl)

;; Local policy: For foo.wrl, we want the last variant,
;; but everything else we'll settle for the first.
(define ((keep! yes) ext)
  (put-mime-types!
   'stomp ext
   (yes (mime-types<-extension ext))))
```

```
((keep! reverse) 'wrl)
(for-each (keep! car) AMBIGUOUS)

(for-each display-ext AMBIGUOUS)
    asn  chemical/x-ncbi-asn1
    wrl  x-world/x-vrml
    tcl  application/x-tcl
     ra  audio/x-pn-realaudio
    spl  application/futuresplash
    m3u  audio/mpegurl
    ent  chemical/x-ncbi-asn1-ascii
    sdf  application/vnd.stardivision.math
     sh  application/x-sh
    csh  application/x-csh
    cpt  application/mac-compactpro
```

Seasoned schemers will note that the same result could have been achieved if *resolve* were allowed to be a general resolution procedure instead of simply a method specifier. Perhaps that feature will be added in the future, and `select-extensions` replaced by `map-mime-types`. We'll see. . .

Index

#

<

A

B

C

E

F

H

I

L

M

N

P

Q

R

S

T

U

W