# SWI-Prolog HTTP support

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: `jan@swi.psy.uva.nl`

March 4, 2003

**Abstract**

This article documents the http-package, a series of libraries for accessing data on HTTP servers as well as provide HTTP server capabilities from SWI-Prolog. Both server and client are modular libraries. The server can be operated from the Unix `inetd` super-daemon as well as as a stand-alone server.

# Contents

# 1 Introduction

The HTTP (HyperText Transfer Protocol) is the W3C standard protocol for transferring information between a web-client (browser) and a web-server. The protocol is a simple *envelope* protocol where standard name/value pairs in the header are used to split the stream into messages and communicate about the connection-status. Many languages have client and or server libraries to deal with the HTTP protocol, making it a suitable candidate for general purpose client-server applications. It is the basis of popular agent protocols such as SOAP and FIPA.

In this document we describe a modular infra-structure to access web-servers from SWI-Prolog and turn Prolog into a web-server. The server code is designed to allow the same 'body' to be used from an interactive server for debugging or providing services from otherwise interactive applications, run the body from an *inetd* super-server or as a CGI script behind a generic web-server.

The design of this module is different from the competing XPCE-based HTTP server located in `http/httpd.pl`. This library intensively uses XPCE functionality to reach its goals. This is not very suitable for CGI or inetd-driven servers due to required X11 connection and much larger footprint.

### Acknowledgements

This work has been carried out under the following projects: GARP, MIA, IBROW and KITS. The following people have pioneered parts of this library and contributed with bug-report and suggestions for improvements: Anjo Anjewierden, Bert Bredeweg, Wouter Jansweijer and Bob Wielinga.

# 2 The HTTP client libraries

This package provides two packages for building HTTP clients. The first, `http/http_open` is a very lightweight library for opening a HTTP URL address as a Prolog stream. It can only deal with the HTTP GET protocol. The second, `http/http_client` is a more advanced library dealing with *keep-alive*, *chunked transfer* and a plug-in mechanism providing conversions based on the MIME content-type.

## 2.1 The `http/http_open` library

The library `http/http_open` provides a very simple mechanism to read data from an HTTP server using the HTTP 1.0 protocol and HTTP GET access method. It defines one predicate:

**http_open**(*+URL, -Stream, +Options*)

Open the data at the HTTP server as a Prolog stream. After this predicate succeeds the data can be read from *Stream*. After completion this stream must be closed using the built-in Prolog predicate `close/1`. *Options* provides additional options:

**timeout**(*+Timeout*)

If provided, set a timeout on the stream using `set_stream/2`. With this option if no new data arrives within *Timeout* seconds the stream raises an exception. Default is to wait forever (`infinite`).

**header(**_+Name, -AtomValue_**)**
: If provided, *AtomValue* is unified with the value of the indicated field in the reply header. *Name* is matched case-insensitive and the underscore (_) matches the hyphen (`-`). Multiple of these options may be provided to extract multiple header fields. If the header is not available *AtomValue* is unified to the empty atom ('').

**size(**_-Size_**)**
: If provided *Size* is unified with the value of the `Content-Length` fields of the reply-header.

**proxy(**_+Host, +Port_**)**
: Use an HTTP proxy to connect to the outside world.

**user_agent(**_+Agent_**)**
: Defines the value of the `User-Agent` field of the HTTP header. Default is `SWI-Prolog (http://www.swi-prolog.org)`.

Here is a simple example:

```
?- http_open('http://www.swi-prolog.org/news.html', In, []),
   copy_stream_data(In, user_output),
   close(In).
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">

<HTML>
<HEAD>
<TITLE>News</TITLE>
</HEAD>
...
```

## 2.2 The `http/http_client` library

The `http/http_client` library provides more powerful access to reading HTTP resources, providing *keep-alive* connections, *chunked* transfer and conversion of the content, such as breaking down *multipart* data, parsing HTML, etc. The library announces itself as providing `HTTP/1.1`.

**http_get(**_+URL, -Reply, +Options_**)**
: Performs a HTTP GET request on the given URL and then reads the reply using `http_read_data/3`. Defined options are:

**connection(**_ConnectionType_**)**
: If `close` (default) a new connection is created for this request and closed after the request has completed. If `'Keep-Alive'` the library checks for an open connection on the requested host and port and re-uses this connection. The connection is left open if the other party confirms the keep-alive and closed otherwise.

**http_version(**_Major-Minor_**)**
: Indicate the HTTP protocol version used for the connection. Default is `1.1`.

**proxy**(*+Host, +Port*)
Use an HTTP proxy to connect to the outside world.

**user_agent**(*+Agent*)
Defines the value of the `User-Agent` field of the HTTP header. Default is `SWI-Prolog (http://www.swi-prolog.org)`.

Remaining options are passed to `http_read_data/3`.

**http_post**(*+URL, +In, -Reply, +Options*)
Performs a HTTP POST request on the given URL. It is equivalent to `http_get/3`, except for providing an *input document*, which is posted using `http_post_data/3`.

**http_read_data**(*+Header, -Data, +Options*)
Read data from an HTTP stream. Normally called from `http_get/3` or `http_post/4`. When dealing with HTTP POST in a server this predicate can be used to retrieve the posted data. *Header* is the parsed header. *Options* is a list of `Name`(*Value*) pairs to guide the translation of the data. The following options are supported:

**to**(*Target*)
Do not try to interpret the data according to the MIME-type, but return it literally according to *Target*, which is one of:

**stream**(*Output*)
Append the data to the given stream, which should be a Prolog stream open for writing. This can be used to return save the data in a (memory-)file, XPCE object, forward it to process using a pipe, etc.

**atom**
Return the result as an atom. Though SWI-Prolog has no limit on the size of atoms and provides atom-garbage collection, this options should be used with care.[1]

**codes**
Return the page as a list of character-codes. This is especially useful for parsing it using grammar rules.

If no `to`(*Target*) option is provided the library tries the registered plug-in conversion filters. If none of these succeed it tries the built-in content-type handlers or returns the content as an atom. The builtin content filters are described below. The provided plug-ins are described in the following sections.

**application/x-www-form-urlencoded**
This is the default encoding mechanism for POST requests issued by a web-browser. It is broken down to a list of *Name* = *Value* terms.

Finally, if all else fails the content is returned as an atom.

[1] Currently atom-garbage collection is activated after the creation of 10,000 atoms.

### 2.2.1 The MIME client plug-in

This plug-in library `http/http_mime_plugin` breaks multipart documents that are recognised by the `Content-Type: multipart/form-data` or `Mime-Version: 1.0` in the header into a list of *Name* = *Value* pairs. This library deals with data from web-forms using the `multipart/form-data` encoding as well as the FIPA agent-protocol messages.

### 2.2.2 The SGML client plug-in

This plug-in library `http/http_sgml_plugin` provides a bridge between the SGML/XML/HTML parser provided by `sgml` and the http client library. After loading this hook the following mime-types are automatically handled by the SGML parser.

**text/html**
: Handed to `sgml` using W3C HTML 4.0 DTD, suppressing and ignoring all HTML syntax errors. *Options* is passed to `load_structure/3`.

**text/xml**
: Handed to `sgml` using dialect `xmlns` (XML + namespaces). *Options* is passed to `load_structure/3`. In particular, `dialect`(*xml*) may be used to suppress namespace handling.

**text/x-sgml**
: Handled to `sgml` using dialect `sgml`. *Options* is passed to `load_structure/3`.

## 3 The HTTP server libraries

The HTTP server library consists of two parts. The first deals with connection management and has three different implementation depending on the desired type of server. The second implements a generic wrapper for decoding the HTTP request, calling user code to handle the request and encode the answer. This design is summarised in figure 1.

The functional body of the user's code is independent from the selected server-type, making it easy to switch between the supported server types. Especially the XPCE-based event-driven server is comfortable for debugging but less suitable for production servers. We start the description with how the user must formulate the functionality of the server.

### 3.1 The 'Body'

The server-body is the code that handles the request and formulates a reply. To facilitate all mentioned setups, the body is driven by `http_wrapper/3`. The goal is called with the parsed request (see section 3.2) as argument and `current_output` set to a temporary buffer. Its task is closely related to the task of a CGI script; it must write a header declaring holding at least the `Content-type` field and a body. Here is a simple body writing the request as an HTML table.

```
reply(Request) :-
        format('Content-type: text/html~n~n', []),
        format('<html>~n', []),
        format('<table border=1>~n'),
```

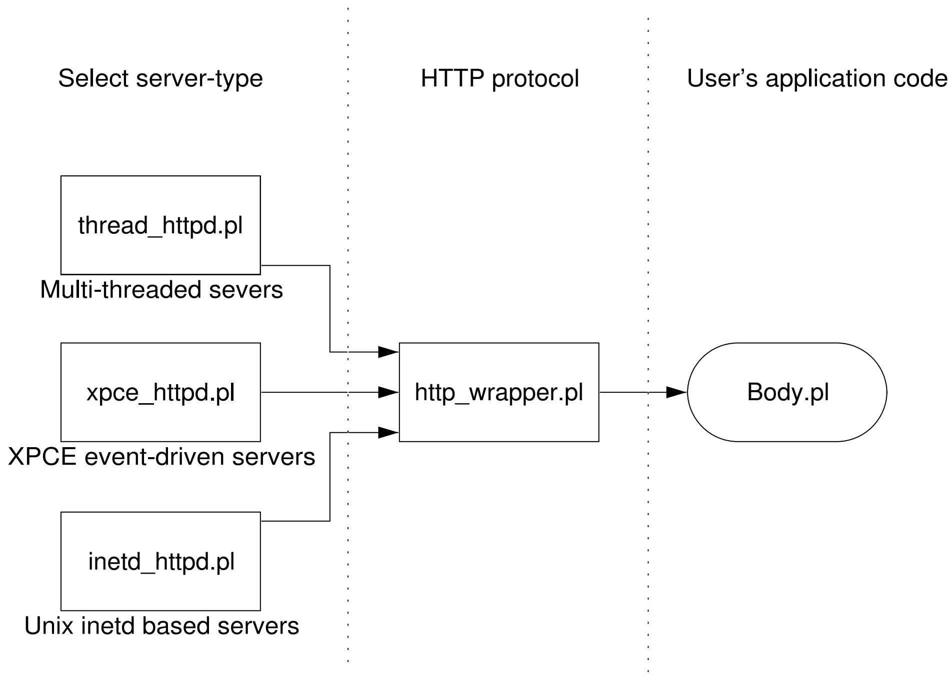


Figure 1: Design of the HTTP server

```
        print_request(Request),
        format('~n</table>~n'),
        format('</html>~n', []).

print_request([]).
print_request([H|T]) :-
        H =.. [Name, Value],
        format('<tr><td>~w<td>~w~n', [Name, Value]),
        print_request(T).
```

## 3.2 Request format

The body-code (see section 3.1) is driven by a *Request*. This request is generated from `http_read_request/2` defined in `http/http_header`.

**http_read_request(***+Stream, -Request***)**
Reads an HTTP request from *Stream* and unify *Request* with the parsed request. *Request* is a list of *Name*(*Value*) elements. It provides a number of predefined elements for the result of parsing the first line of the request, followed by the additional request parameters. The predefined fields are:

**input(***Stream***)**
The *Stream* is passed along, allowing to read more data or requests from the same stream. This field is always present.

**method(***Method***)**
: *Method* is one of `get`, `put` or `post`. This field is present if the header has been parsed successfully.

**path(***Path***)**
: Path associated to the request. This field is always present.

**search(***ListOfNameValue***)**
: Search-specification of URI. This is the part after the `?`, normally used to transfer data from HTML forms that use the '`GET`' protocol. In the URL it consists of a www-form-encoded list of *Name*=*Value* pairs. This is mapped to a list of Prolog *Name*=*Value* terms with decoded names and values. This field is only present if the location contains a search-specification.

**http_version(***Major-Minor***)**
: If the first line contains the `HTTP/`*Major.Minor* version indicator this element indicate the HTTP version of the peer. Otherwise this field is not present.

If the first line of the request is tagged with `HTTP/`*Major.Minor*, `http_read_request/2` reads all input upto the first blank line. This header consists of *Name*:*Value* fields. Each such field appears as a term *Name*(*Value*) in the *Request*, where *Name* is canonised for use with Prolog. Canonisation implies that the *Name* is converted to lower case and all occurrences of the `-` are replaced by `_`. The value for the `Content-length` fields is translated into an integer.

Here is an example:

```
?- http_read_request(user, X).
|: GET /mydb?class=person HTTP/1.0
|: Host: gollem
|:
X = [ input(user),
      method(get),
      search([ class = person
             ]),
      path('/mydb'),
      http_version(1-0),
      host(gollem)
    ].
```

### 3.2.1 Handling POST requests

Where the HTTP `GET` operation is intended to get a document, using a *path* and possibly some additional search information, the `POST` operation is intended to hand potentially large amounts of data to the server for processing.

The *Request* parameter above contains the term `method`(*post*). The data posted is left on the input stream that is available through the term `input`(*Stream*) from the *Request* header. This data can be read using `http_read_data/3` from the HTTP client library. Here is a demo implementation simply returning the parsed pasted data as plain http://db.cwi.nl/projecten/project.php4?prjnr=129text (assuming `pp/1` pretty-prints the data).

```
reply(Request) :-
        member(method(post), Request), !,
        http_read_data(Request, Data, []),
        format('Content-type: text/plain~n~n', []),
        pp(Data).
```

If the POST is initiated from a browser, content-type is generally either `application/x-www-form-urlencoded` or `multipart/form-data`. The latter is broken down automatically if the plug-in `http/http_mime_plugin` is loaded.

### 3.3 Running the server

The functionality of the server should be defined in one Prolog file (of course this file is allowed to load other files). Depending on the wanted server setup this 'body' is wrapped into a small Prolog file combining the body with the appropriate server interface. There are three supported server-setups:

- *Using* `xpce_httpd` *for an event-driven server*
  This approach provides a single-threaded event-driven application. The clients talk to XPCE sockets that collect an HTTP request. The server infra-structure can talk to multiple clients simultaneously, but once a request is complete the wrappers call the user's goal and blocks all further activity until the request is handled. Requests from multiple clients are thus fully serialised in one Prolog process.

  This server setup is very suitable for debugging as well as embedded server in simple applications in a fairly controlled environment.

- *Using* `thread_httpd` *for a multi-threaded server*
  This server exploits the multi-threaded version of SWI-Prolog, running the users body code parallel from a pool of worker threads. As it avoids the state engine and copying required in the event-driven server it is generally faster and capable to handle multiple requests concurrently.

  This server is a harder to debug due to the involved threading. It can provide fast communication to multiple clients and can be used for more demanding embedded servers, such as agent platforms.

- *Using* `inetd_httpd` *for server-per-client*
  In this setup the Unix `inetd` user-daemon is used to initialise a server for each connection. This approach is especially suitable for servers that have a limited startup-time. In this setup a crashing client does not influence other requests.

  This server is very hard to debug as the server is not connected to the user environment. It provides a robust implementation for servers that can be started quickly.

### 3.3.1 Common server interface options

All the server interfaces provide `http_server`(*:Goal, +Options*) to create the server. The list of options differ, but the servers share common options:

**port(*?Port*)**
: Specify the port to listen to for stand-alone servers. *Port* is either an integer or unbound. If unbound, it is unified to the selected free port.

**after(*:Goal*)**
: Specify a goal to be run on the query just like the first argument of `http_server/2`. This goal however is started *after* the request has been answered. It is called using `call`(*Goal, Request*). This extension was added to support the FIPA-HTTP protocol, which issues HTTP POST requests on the server. The server answers these requests with an empty document before starting processing. The `after`-option is used for the processing:

```
:- http_server(reply, [after(action), ...]).

reply(Request) :-
        format('Content-type: text/plain\r\n\r\n').

action(Request) :-
        <start agent work on request>
```

### 3.3.2 From an interactive Prolog session using XPCE

The `http/xpce_httpd.pl` provides the infrastructure to manage multiple clients with an event-driven control-structure. This version can be started from an interactive Prolog session, providing a comfortable infra-structure to debug the body of your server. It also allows the combination of an (XPCE-based) GUI with web-technology in one application.

**http_server(*:Goal, +Options*)**
: Create an instance of *interactive_httpd*. *Options* must provide the `port`(*?Port*) option to specify the port the server should listen to. If *Port* is unbound an arbitrary free port is selected and *Port* is unified to this port-number. The only other option provided is the `after`(*:Goal*) option.

The file `demo_xpce` gives a typical example of this wrapper, assuming `demo_body` defines the predicate `reply/1`.

```
:- use_module(xpce_httpd).
:- use_module(demo_body).

server(Port) :-
        http_server(reply, Port, []).
```

The created server opens a server socket at the selected address and waits for incoming connections. On each accepted connection it collects input until an HTTP request is complete. Then it opens an input stream on the collected data and using the output stream directed to the XPCE *socket* it calls `http_wrapper/3`. This approach is fundamentally different compared to the other approaches:

- *Server can handle multiple connections*
  When *inetd* will start a server for each *client*, and CGI starts a server for each *request*, this approach starts a single server handling multiple clients.

- *Requests are serialised*
  All calls to *Goal* are fully serialised, processing on behalf of a new client can only start after all previous requests are answered. This easier and quite acceptable if the server is mostly inactive and requests take not very long to process.

- *Lifetime of the server*
  The server lives as long as Prolog runs.

### 3.3.3 Multi-threaded Prolog

The `http/thread_httpd.pl` provides the infrastructure to manage multiple clients using a pool of *worker-threads*. This realises a popular server design, also seen in SUN JavaBeans and Microsoft .NET. As a single persistent server process maintains communication to all clients startup time is not an important issue and the server can easily maintain state-information for all clients.

**http_server**(*:Goal, +Options*)
: Create the server. *Options* must provide the `port`(*?Port*) option to specify the port the server should listen to. If *Port* is unbound an arbitrary free port is selected and *Port* is unified to this port-number. The server consists of a small Prolog thread accepting new connection on *Port* and dispatching these to a pool of workers. Defined *Options* are:

  **port**(*?Port*)
  : Port the server should listen to. If unbound *Port* is unified with the selected free port.

  **workers**(*+N*)
  : Defines the number of worker threads in the pool. Default is to use *two* workers. Choosing the optimal value for best performance is a difficult task depending on the number of CPUs in your system and how much resources are required for processing a request. Too high numbers makes your system switch too often between threads or even swap if there is not enough memory to keep all threads in memory, while a too low number causes clients to wait unnecessary for other clients to complete. See also `http_workers/2`.

  **timeout**(*+SecondsOrInfinite*)
  : Determines the maximum period of inactivity handling a request. If no data arrives within the specified time since the last data arrived the connection raises an exception, the worker discards the client and returns to the pool-queue for a new client. Default is `infinite`, making each worker wait forever for a request

to complete. Without a timeout, a worker may wait forever on an a client that doesn't complete its request.

**local(** *+KBytes* **)**
Size of the local-stack for the workers. Default is taken from the commandline option.

**global(** *+KBytes* **)**
Size of the global-stack for the workers. Default is taken from the commandline option.

**trail(** *+KBytes* **)**
Size of the trail-stack for the workers. Default is taken from the commandline option.

**after(** *:Goal* **)**
After replying a request, execute *Goal* providing the request as argument.

**http_current_server(** *?:Goal, ?Port* **)**
Query the running servers. Note that `http_server/3` can be called multiple times to create multiple servers on different ports.

**http_workers(** *:Port, ?Workers* **)**
Query or manipulate the number of workers of the server identified by *Port*. If *Workers* is unbound it is unified with the number of running servers. If it is an integer greater than the current size of the worker pool new workers are created with the same specification as the running workers. If the number is less than the current size of the worker pool, this predicate inserts a number of 'quit' requests in the queue, discarding the excess workers as they finish their jobs (i.e. no worker is abandoned while serving a client).

This can be used to tune the number of workers for performance. Another possible application is to reduce the pool to one worker to facilitate easier debugging.

#### 3.3.4 From (Unix) inetd

All modern Unix systems handle a large number of the services they run through the super-server *inetd*. This program reads `/etc/inetd.conf` and opens server-sockets on all ports defined in this file. As a request comes in it accepts it and starts the associated server such that standard I/O refers to the socket. This approach has several advantages:

- *Simplification of servers*
  Servers don't have to know about sockets and -operations.
- *Centralised authorisation*
  Using *tcpwrappers* simple and effective firewalling of all services is realised.
- *Automatic start and monitor*
  The inetd automatically starts the server 'just-in-time' and starts additional servers or restarts a crashed server according to the specifications.

The very small generic script for handling inetd based connections is in `inetd_httpd`, defining `http_server/1`:

**http_server(**:*Goal, +Options***)**

Initialises and runs `http_wrapper/3` in a loop until failure or end-of-file. This server does not support the *Port* option as the port is specified with the `inetd` configuration. The only supported option is *After*.

Here is the example from `demo_inetd`

```
#!/usr/bin/pl -t main -q -f
:- use_module(demo_body).
:- use_module(inetd_httpd).

main :-
        http_server(reply).
```

With the above file installed in `/home/jan/plhttp/demo_inetd`, the following line in `/etc/inetd` enables the server at port 4001 guarded by *tcpwrappers*. After modifying inetd, send the daemon the `HUP` signal to make it reload its configuration. For more information, please check `inetd.conf(5)`.

```
4001 stream tcp nowait nobody /usr/sbin/tcpd /home/jan/plhttp/demo_inetd
```

### 3.3.5 MS-Windows

There are rumours that *inetd* has been ported to Windows.

### 3.3.6 As CGI script

To be done.

## 3.4 The wrapper library

The body is called by the module `http/http_wrapper.pl`. This module realises the communication between the I/O streams and the body described in section 3.1. The interface is realised by `http_wrapper/3`:

**http_wrapper(**:*Goal, +In, +Out, -Connection, +Options***)**

Handle an HTTP request where *In* is an input stream from the client, *Out* is an output stream to the client and *Goal* defines the goal realising the body. *Connection* is unified to `'Keep-alive'` if both ends of the connection want to continue the connection or `close` if either side wishes to close the connection. The only option provided is `request`(-*Request*), providing the executed request to the caller.

This predicate reads an HTTP request-header from *In*, redirects current output to a memory file and then runs `call(Goal, Request)`, watching for exceptions and failure. If *Goal* executes successfully it generates a complete reply from the created output. Otherwise it generates an HTTP server error with additional context information derived from the exception.

### 3.5 The `http/html_write` library

Producing output for the web in the form of an HTML document is a requirement for many Prolog programs. Just using `format/2` is satisfactory as it leads to poorly readable programs generating poor HTML. This library is based on using DCG rules.

The `http/html_write` structures the generation of HTML from a program. It is an extensible library, providing a *DCG* framework for generating legal HTML under (Prolog) program control. It is especially useful for the generation of structured pages (e.g. tables) from Prolog data structures.

The normal way to use this library is through the DCG `html/1`. This grammar-rule provides the central translation from a structured term with embedded calls to additional translation rules to a list of atoms that can then be printed using `print_html/[1,2]`.

**html**(*:Spec*) `-->`

http://db.cwi.nl/projecten/project.php4?prjnr=129The DCG rule `html/1` is the main predicate of this library. It translates the specification for an HTML page into a list of atoms that can be written to a stream using `print_html/[1,2]`. The expansion rules of this predicate may be extended by defining the multifile DCG html_write:`expand/1`. *Spec* is either a single specification or a list of single specifications. Using nested lists is not allowed to avoid ambiguity caused by the atom `[]`

- *Atomic data*
  Atomic data is quoted using the `html_quoted/1` DCG.
- Fmt - Args
  *Fmt* and *Args* are used as format-specification and argument list to `sformat/3`. The result is quoted and added to the output list.
- `\`
  Escape sequence to add atoms directly to the output list. This can be used to embed external HTML code.
- `\`
  Invoke the grammar rule *Term* in the calling module. This is the common mechanism to realise abstraction and modularisation in generating HTML.
- Module:Term
  Invoke the grammar rule ⟨*Module*⟩:⟨*Term*⟩. This is similar to `\` but allows for invoking grammar rules in external packages.
- *&(Entity)*
  Emit `&`⟨*Entity*⟩`;`.
- `Tag`*(Content)*
  Emit HTML element *Tag* using *Content* and no attributes. *Content* is handled to `html/1`. See section 3.5.3 for details on the automatically generated layout.
- `Tag`*(Attributes, Content)*
  Emit HTML element *Tag* using *Attributes* and *Content*. *Attributes* is either a single attribute of a list of attributes. Each attributes is of the format `Name(`*Value*`)` or `Name(`*Value*`)`.

**page**(*:HeadContent, :BodyContent*) `-->`

The DCG rule `page/2` generated a complete page, including the SGML `DOCTYPE` declaration. *HeadContent* are elements to be placed in the `head` element and *BodyContent* are elements to be placed in the `body` element.

http://db.cwi.nl/projecten/project.php4?prjnr=129To achieve common style (background, page header and footer), it is possible to define DCG rules `head/1` and/or `body/1`. The `page/1` rule checks for the definition of these DCG rules in the module it is called from as well as in the `user` module. If no definition is found, it creates a head with only the *HeadContent* (note that the `title` is obligatory) and a `body` with `bgcolor` set to `white` and the provided *BodyContent*.

Note that further customisation is easily achieved using `html/1` directly as `page/2` is (besides handling the hooks) defined as:

```
page(Head, Body) -->
        html([ \['<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 4.0//EN">\n'],
               html([ head(Head),
                      body(bgcolor(white), Body)
                    ])
             ]).
```

**page**(*:Contents*) `-->`

This version of the `page/[1,2]` only gives you the SGML `DOCTYPE` and the `HTML` element. *Contents* is used to generate both the head and body of the page.

**html_begin**(*+Begin*) `-->`

Just open the given element. *Begin* is either an atom or a compound term, In the latter case the arguments are used as arguments to the begin-tag. Some examples:

```
html_begin(table)
html_begin(table(border(2), align(center)))
```

This predicate provides an alternative to using the `\` syntax in the `html/1` specification. The following two fragments are the same. The preferred solution depends on your preferences as well as whether the specification is generated or entered by the programmer.

```
table(Rows) -->
        html(table([border(1), align(center), width('80%')],
                   [ \table_header,
                     \table_rows(Rows)
                   ])).

% or

table(Rows) -->
```

```
html_begin(table(border(1), align(center), width('80%'))),
table_header,
table_rows,
html_end(table).
```

**html_end**(*+End*) `-->`
End an element. See `html_begin/1` for details.

### 3.5.1 Emitting HTML documents

The `html/1` grammar rules translates a specification into a list of atoms and layout instructions. Currently the layout instructions are terms of the format `nl`(*N*), requesting at least *N* newlines. Multiple consequtive `nl`(*1*) terms are combined to an atom containing the maximum of the requested number of newline characters.

To simplify handing the data to a client or storing it into a file, the following predicates are available from this library:

**print_html**(*+List*)
Print the token list to the Prolog current output stream.

**print_html**(*+Stream, +List*)
Print the token list to the specified output stream

**html_print_length**(*+List, -Length*)
When calling `html_print/[1,2]` on *List*, *Length* characters will be produced. Knowing the length is needed to provide the `Content-length` field of an HTTP reply-header.

### 3.5.2 Adding rules for `html/1`

In some cases it is practical to extend the translations imposed by `html/1`. When using XPCE for example, it is comfortable to be able defining default translation to HTML for objects. We also used this technique to define translation rules for the output of the SWI-Prolog `sgml` package.

The `html/1` rule first calls the multifile ruleset html_write:`expand/1`. The other predicates contain commonly rules for defining new rules.

**html_write:expand**(*+Spec*) `-->`
Hook to add additional translationrules for `html/1`.

**html_quoted**(*+Atom*) `-->`
Emit the text in *Atom*, inserting entity-references for the SGML special characters `<&>`.

**html_quoted_attribute**(*+Atom*) `-->`
Emit the text in *Atom* suitable for use as an SGML attribute, inserting entity-references for the SGML special characters `<&>'"`.

### 3.5.3 Generating layout

Though not strictly necessary, the library attempts to generate reasonable layout in SGML output. It does this only by inserting newlines before and after tags. It does this on the basis of the multifile predicate html_write:`layout/3`

**html_write:layout(***+Tag, -Open, -Close***)**

Specify the layout conventions for the element *Tag*, which is a lowercase atom. *Open* is a term *Pre-Post*. It defines that the element should have at least *Pre* newline characters before and *Post* after the tag. The *Close* specification is similar, but in addition allows for the atom `-`, requesting the output generator to omit the close-tag altogether or `empty`, telling the library that the element has declared empty content. In this case the close-tag is not emitted either, but in addition `html/1` interprets *Arg* in `Tag`(*Arg*) as a list of attributes rather than the content.

A tag that does not appear in this table is emitted without additional layout. See also `print_html/[1,2]`. Please consult the library source for examples.

### 3.5.4 Examples

In the following example we will generate a table of Prolog predicates we find from the SWI-Prolog help system based on a keyword. The primary database is defined by the predicate `predicate/5` We will make hyperlinks for the predicates pointing to their documentation.

```
html_apropos(Kwd) :-
        findall(Pred, apropos_predicate(Kwd, Pred), Matches),
        phrase(apropos_page(Kwd, Matches), Tokens),
        print_html(Tokens).

%       emit page with title, header and table of matches

apropos_page(Kwd, Matches) -->
        page([ title(['Predicates for ', Kwd])
             ],
             [ h2(align(center),
                  ['Predicates for ', Kwd]),
               table([ align(center),
                       border(1),
                       width('80%')
                     ],
                     [ tr([ th('Predicate'),
                            th('Summary')
                          ])
                     | \apropos_rows(Matches)
                     ])
             ]).

%       emit the rows for the body of the table.
```

```
apropos_rows([]) -->
        [].
apropos_rows([pred(Name, Arity, Summary)|T]) -->
        html([ tr([ td(\predref(Name/Arity)),
                    td(em(Summary))
                  ])
             ]),
        apropos_rows(T).

%       predref(Name/Arity)
%
%       Emit Name/Arity as a hyperlink to
%
%               /cgi-bin/plman?name=Name&arity=Arity
%
%       we must do form-encoding for the name as it may contain illegal
%       characters.  www_form_encode/2 is defined in library(url).

predref(Name/Arity) -->
        { www_form_encode(Name, Encoded),
          sformat(Href, '/cgi-bin/plman?name=~w&arity=~w',
                  [Encoded, Arity])
        },
        html(a(href(Href), [Name, /, Arity])).

%       Find predicates from a keyword. '$apropos_match' is an internal
%       undocumented predicate.

apropos_predicate(Pattern, pred(Name, Arity, Summary)) :-
        predicate(Name, Arity, Summary, _, _),
        (   '$apropos_match'(Pattern, Name)
        ->  true
        ;   '$apropos_match'(Pattern, Summary)
        ).
```

### 3.5.5 Remarks on the `http/html_write` library

This library is the result of various attempts to reach at a more satisfactory and Prolog-minded way to produce HTML text from a program. We have been using Prolog for the generation of web pages in a number of projects. Just using `format/2` never was a real option, generating error-prone HTML from clumsy syntax. We started with a layour on top of format, keeping track of the current nesting and thus always capable of properly closing the environment.

DCG based translation however naturally exploits Prologs term-rewriting primitives. If generation fails for whatever reason it is easy to produce an alternative document (for example holding an error message).

The approach presented in this library has been used in combination with `http/httpd` in three projects: viewing RDF in a browser, selecting fragments from an analysed document and presenting parts of the XPCE documentation using a browser. It has proven to be able to deal with generating pages quickly and comfortably.

In a future version we will probably define a `goal_expansion/2` to do compile-time optimisation of the library. Quotation of known text and invokation of sub-rules using the \ and ⟨*Module*⟩:⟨*RuleSet*⟩ operators are costly operations in the analysis that can be done at compile-time.

# 4 Security

Writing servers is an inherently dangerous job that should be carried out with some considerations. You have basically started a program on a public terminal and invited strangers to use it. When using the interactive server or inetd based server the server runs under your privileges. Using CGI scripted it runs with the privileges of your web-server. Though it should not be possible to fatally compromise a Unix machine using user privileges, getting unconstrained access to the system is highly undesirable.

Symbolic languages have an additional handicap in their inherent possibilities to modify the running program and dynamically create goals (this also applies to the popular perl and java scripting languages). Here are some guidelines.

- *Check your input*
  Hardly anything can go wrong if you check the validity of query-arguments before formulating an answer.

- *Check filenames*
  If part of the query consists of filenames or directories, check them. This also applies to files you only read. Passing names as `/etc/passwd`, but also `../../../../../etc/passwd` are tried by experienced hackers to learn about the system they want to attack. So, expand provided names using `absolute_file_name/[2,3]` and verify they are inside a folder reserved for the server. Avoid symbolic links from this subtree to the outside world. The example below checks validity of filenames. The first call ensures proper canonisation of the paths to avoid an mismatch due to symbolic links or other filesystem ambiguities.

```
check_file(File) :-
        absolute_file_name('/path/to/reserved/area', Reserved),
        absolute_file_name(File, Tried),
        atom_concat(Reserved, _, Tried).
```

- *Check scripts*
  Should input in any way activate external scripts using `shell/1` or `open(pipe(Command), ...)`, verify the argument once more.

- *Check meta-calling*
  *The* attractive situation for you and your attacker is below:

```
reply(Query) :-
        member(search(Args), Query),
        member(action=Action, Query),
        member(arg=Arg, Query),
        call(Action, Arg).             % NEVER DO THIS
```

All your attacker has to do is specify *Action* as `shell` and *Arg* as `/bin/sh` and he has an uncontrolled shell!

## 5 Status

The current library has been developed and tested in a number of internal and funded projects at the SWI department of the University of Amsterdam. With this release we hope to streamline deployment within these projects as well as let other profit from the possibilities to use Prolog directly as a web-server.

This library is by no means complete and you are free to extend it. Partially or completely lacking are notably session management and authorisation.