# `html.pl`: A Simple HTML Package for Prolog and CLP Systems

*Description and User's Manual*

(Version 96.1.1)

D. Cabeza M. Hermenegildo *

15 Mar 1996

**Abstract**

We describe a simple, public domain, HTML package for LP/CLP systems. The package allows generating HTML documents easily from LP/CLP systems, including HTML *forms*. It also provides facilities for parsing the input provided by HTML forms, as well as for creating standalone form handlers. The purpose of this document is to serve as a user's manual as well as a short description of the capabilities of the package. The package was originally developed for SICStus Prolog and the UPM &-Prolog/CIAO systems, but has been adapted to a number of popular LP/CLP systems.

The document is also a WWW/HTML primer, containing sufficient information for developing medium complexity WWW applications in Prolog and other LP and CLP languages.

## 1 Introduction

Prolog, and its constraint based extensions, are excellent tools for developing knowledge-based applications. It is often useful to interface such applica-

*Computer Science Department, Technical University of Madrid (UPM), Spain, `dcabeza@fi.upm.es`, `herme@fi.upm.es`.

tions to one of the main on-line information exchange forums, the World Wide Web (WWW). This Prolog HTML package aims at providing a set of library procedures which should be useful in this process. The package allows generating documents in HTML format, the native format of WWW documents, easily from Prolog, including HTML *forms*. It also provides facilities for parsing the input provided by HTML forms (independently of whether they have been produced by a Prolog program or not), and for producing stand-alone form handlers. The purpose of this document is to serve as a user's manual as well as a short description of the capabilities of the package. The package was originally developed for SICStus Prolog [2] and the UPM &-Prolog [5] and CIAO systems [4, 3], but has also been adapted to other popular LP/CLP systems (see the `Makefile` for details). It should be straightforward to adapt it to other Prolog and CLP systems.

If you find this package useful and develop an application with it, we would appreciate if you mention our work by citing the user's manual and/or by putting a pointer to its location in our WWW pages (`http://www.clip.dia.fi.upm.es/miscdocs/html_pl/html_pl.html`).

# 2 Getting and Installing `html.pl`

The latest version of the Prolog HTML package is available via anonymous ftp from the CLIP ftp server.[1] To install simply uncompress and untar the distribution in a convenient directory. Then edit the options in `Makefile` to suit your system. In particular, copy the files in the `images` directory to an accessible directory in your server and change

```
ICONDIR=http://www.clip.dia.fi.upm.es/images/
```

to point to this directory (if you cannot do that, bear in mind that these images will have to travel from Spain, Europe!). Also, if you will be generating `cgi-bin` executables (as explained in section 4.2, only available in some systems) change

```
CGIBINDIR=/usr/local/etc/httpd/cgi-bin/
```

to point to a `cgi-bin`-type directory where you have write access. Finally, change `SYSTEM` to the Prolog system you will be using (if your system is not

[1] ftp://clip.dia.fi.upm.es/pub/software

listed, perhaps you can make a compatibility file `c_*.pl` for it, providing the necessary support predicates. If you do so, please send us the changes!). Now, run "`make html.pl`". If your OS does not have "make", then manually produce `html.pl` from `html0.pl` and the appropriate `c_*.pl` compatibility file, changing `<ICONDIR>` and `<CGIBINDIR>` in the code to the correct values.

Versions of this manual in "`.dvi`", "`.ps`", and "`.html`" formats are provided in the distribution in the `doc` directory. The latest version of this manual can be accessed on-line from

`http://www.clip.dia.fi.upm.es/miscdocs/html_pl/html_pl.html`

# 3 Contents of the Distribution

`README`, `COPYING.LIB` Read these files before using the package.

`CHANGES` History of the changes between versions.

`Makefile` Makefile for producing `html.pl`. Must be edited to suit your system.

`c_`*system*`.pl` Compatibility file for system *system*.

`html0.pl` Main source code from which `html.pl` is made by `make`.

`html.pl` This is the main library file which should be loaded into the application. The default version in the distribution is for &-Prolog/CIAO, change `Makefile` options and run `make` to make a version for your system.

`doc` Directory containing the documentation.

- `html_pl.tex` This manual, in LaTeX format.
- `html_pl.ps` This manual, in postscript format.
- `html_pl/html_pl.html` This manual, in html format. Needs also the other files in this directory.

`examples` Directory containing examples.

`images` Directory containing standard images used in the package.

# 4 Using the Package

The package comprises three basic building blocks:

- Facilities for generating HTML documents (including forms).
- Facilities for parsing input from HTML forms.
- Facilities for producing stand-alone form handlers.

The functionality and predicates of each of these blocks are described in the following sections.

## 4.1 Predicates for Generating HTML Documents

`output_html(F)` Accepts in $F$ an HTML term (or a list of HTML terms) and sends to the standard output the text which is the rendering of the term(s) in HTML format.

`html_term(F,L,T)` Accepts in $F$ an HTML term and produces in $L$ a list of atoms which are the rendering of the term in HTML format. $T$ is the tail of this list and is usually `[]`. Used by `output_html/2`.

An *HTML term* is a Prolog term in which certain atoms and structures represent special functionality at the HTML level. It can be recursively a list of HTML terms. The following are legal HTML terms:

```
hello
[hello, world]
['This is an ', em('HTML'), ' term']
```

`html_term/3` in general leaves atoms in HTML terms unchanged, but translates structures into the corresponding format in HTML, applying then `html_term/2` recursively to their arguments. For example, given the following code:

```
:- ['/usr/local/lib/prolog/html.pl'].

main :-
  tell('html_simple.html'),
  icon_address(clip,ClipLogo),
```

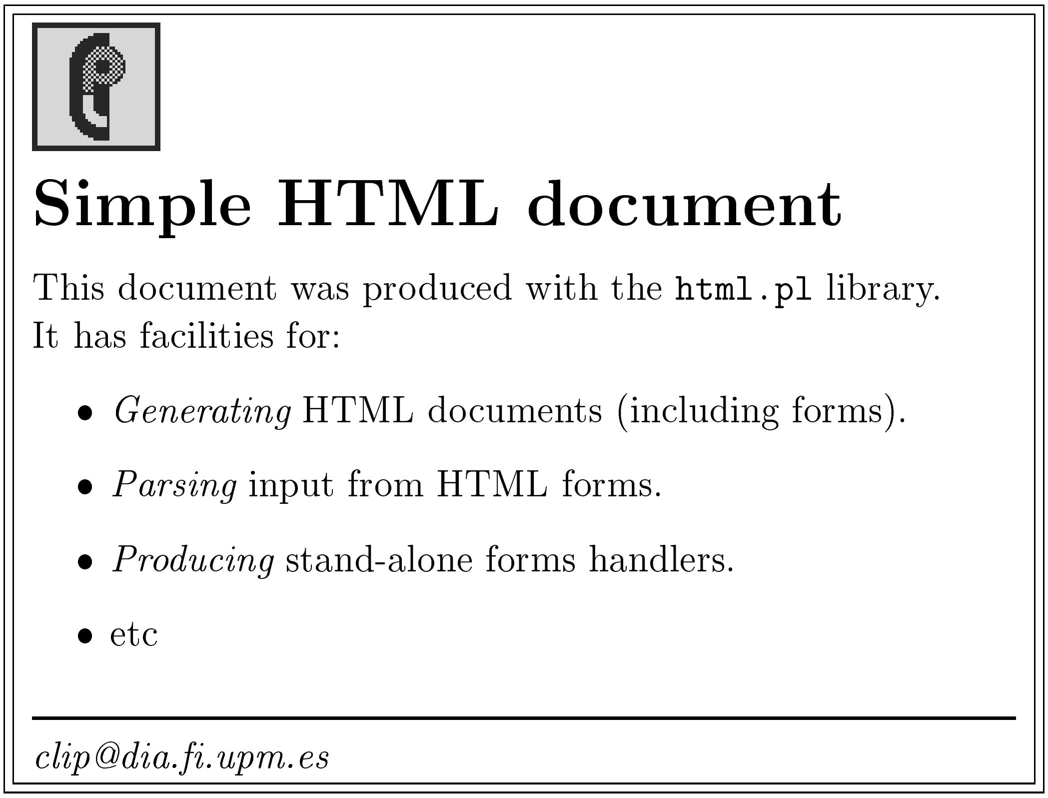
```
output_html([
  start,
  title('Simple HTML document generated from Prolog'),
  image(ClipLogo), nl,
  heading(1,'Simple HTML document'),
  'This document was produced with the ',tt('html.pl'),' library.', \\,
  'It has facilities for:',
  itemize([
    [em('Generating'),' HTML documents (including forms).'],
    [em('Parsing'),' input from HTML forms.'],
    [em('Producing'),' stand-alone forms handlers.'],
    etc
    ]),
  --,
  address('clip@dia.fi.upm.es'),
  end
]),
told.
```

Loading this code and calling `main/0` produces the file `html_simple.html`, the contents of which are:

```
<html><title>Simple HTML document generated from Prolog</title>
<img src="http://www.clip.dia.fi.upm.es/images/clip.gif">
<h1>Simple HTML document</h1>
This document was produced with the <tt>html.pl</tt> library.<br>
It has facilities for:<ul>
<li> <em>Generating</em> HTML documents (including forms).
<li> <em>Parsing</em> input from HTML forms.
<li> <em>Producing</em> stand-alone forms handlers.
<li> etc
</ul>
<hr>
<address>clip@dia.fi.upm.es</address></html>
```

Visualizing this file with a WWW browser produces output similar to figure 1 (the best thing is to try it out with your favorite browser).

As was shown with the previous example, HTML terms may contain logic variables, provided they are instantiated before the term is translated or output. This allows creating documents piecemeal, backpatching of references in documents, etc.



# Simple HTML document

This document was produced with the `html.pl` library.
It has facilities for:

- *Generating* HTML documents (including forms).
- *Parsing* input from HTML forms.
- *Producing* stand-alone forms handlers.
- etc

---

*clip@dia.fi.upm.es*

Figure 1: Possible visualization of `html_simple.html`

In the following sections we list the meaning of the various Prolog structures that represent special functionality at the HTML level. Any atom not included here is assumed to be normal text and will be passed through to the HTML document. These sections are included for reference, but in any case the source file is clear enough (perhaps more than this description). You can also examine section 5 to see examples.

#### 4.1.1 General Structures

Basically, HTML has two kinds of components: HTML *elements* and HTML *environments*. An HTML element has the form `<NAME` *Attributes* `>` were `NAME` is the name of the element and *Attributes* is a (possibly empty) sequence of attributes, each of them being either an attribute name or an attribute assignment as `name="Value"`.

An HTML environment has the form `<NAME` *Attributes* `>` *Text* `</NAME>`

were `NAME` is the name of the environment an *Attributes* has the same form as before.

The general Prolog structures that represent these two HTML constructions are:

$\boxed{Name\$Atts}$ (This is a '`$`' binary operator.) Represents an HTML element of name *Name* and attributes *Atts*, were *Atts* is a (possibly empty) list of attributes, each of them being either an atom or an structure *name*=*value*. For example, the term

```
img$[src='images/map.gif',alt='A map',ismap]
```

is translated into the HTML source

```
<img src="images/map.gif" alt="A map" ismap>
```

Note that HTML is not case-sensitive, so we can use lower-case atoms.

$\boxed{name(Text)}$ (This is a term with functor *name*/1 and argument *Text*) Represents an HTML environment of name *name* and included text *Text*. For example, the term

```
address('clip@dia.fi.upm.es')
```

is translated into the HTML source

```
<address>clip@dia.fi.upm.es</address>
```

$\boxed{name(Atts, Text)}$ (This is a term with functor *name*/2 and arguments *Atts* and *Text*) Represents an HTML environment of name *name*, attributes *Atts* and included text *Text*. For example, the term

```
a([href='http://www.clip.dia.fi.upm.es/'],'Clip home')
```

represents the HTML source

```
<a href="http://www.clip.dia.fi.upm.es/">Clip home</a>
```

In principle, (almost) any HTML construction can be represented with these structures[2], but for the sake of simplifying HTML creation, more specific structures were designed.

[2] A notable exception are comments.

### 4.1.2 Specific Structures

Most of the names of the structures listed here have been fashioned after their LaTeX[6] equivalent. New structures can be defined by means of the `html_expansion/2` predicate (see section 4.1.4). Note that these structures take priority over the above ones.

`start` Used at the beginning of a document (translates to `<html>`).

`end` Used at the end of a document (translates to `</html>`).

`--` Produces a horizontal rule (translates to `<hr>`).

`\\` Produces a line break (translates to `<br>`).

`$` Produces a paragraph break (translates to `<p>`).

`comment(`*Comment*`)` Used to insert an HTML comment (translates to `<!--` *Comment* `-->`).

`title(`*Title*`)` Used to give a title to a document (translates to a `<title>` environment).

`image(`*Addr*`)` Used to include an image of address (URL) *Addr* (translates to an `<img>` element).

`image(`*URL*`,`*Atts*`)` Idem with the list of attributes *Atts*.

`ref(`*Addr*`,`*Text*`)` Produces a hypertext link, *Addr* is the URL of the referenced resource, *Text* is the text of the reference (translates to `<a href="`*Addr*`">`*Text*`</a>`).

`label(`*Label*`,`*Text*`)` Labels *Text* as a target destination with label *Label* (translates to `<a name="`*Label*`">`*Text*`</a>`).

`heading(`*N*`,`*Text*`)` Produces a heading of level *N* ($1 \leq N \leq 6$), *Text* is the text to be used as heading (translates to a `<h`*N*`>` environment).

`itemize(`*Items*`)` Produces a list of bulleted items, *Items* is a list of corresponding HTML terms (translates to a `<ul>` environment).

$\boxed{\texttt{nice\_itemize(}\mathit{Items}\texttt{)}}$ Same as above, but uses nicer bullets. The bullet used can be changed through the predicate `icon_address/2`. (See the source file for details.)

$\boxed{\texttt{enumerate(}\mathit{Items}\texttt{)}}$ Produces a list of numbered items, *Items* is a list of corresponding HTML terms (translates to an `<ol>` environment).

$\boxed{\texttt{description(}\mathit{Defs}\texttt{)}}$ Produces a list of defined items, *Defs* is a list whose elements are definitions, each of them being a Prolog sequence (composed by `','`/2 operators). The last element of the sequence is the definition, the other (if any) are the defined terms (translates to an `<dl>` environment). Section 5 shows an example that should clarify this point.

$\boxed{\texttt{preformatted(}\mathit{Text}\texttt{)}}$ Used to include preformatted text, *Text* is a list of HTML terms, each element of the list being a line of the resulting document (translates to a `<pre>` environment).

$\boxed{\texttt{verbatim(}\mathit{Text}\texttt{)}}$ Used to include text verbatim, special HTML characters (`<,>,&,"`) are translated into its quoted HTML equivalent.

$\boxed{\texttt{nl}}$ Used to include a newline in the HTML source (just to improve human readability).

$\boxed{\texttt{begin(}\mathit{Tag}\texttt{,}\mathit{Atts}\texttt{)}}$ It translates to the start of an HTML environment of name *Tag* and attributes *Atts*. There exists also a `begin(`*Tag*`)` structure. Useful, in conjunction with the next structure, when including in a document output generated by an existing piece of code (e.g. *Tag* = `pre`). Its use is otherwise discouraged.

$\boxed{\texttt{end(}\mathit{Tag}\texttt{)}}$ Translates to the end of an HTML environment of name *Tag*.

### 4.1.3 Specific Structures for Forms

HTML forms are HTML documents (or parts of HTML documents) which, when accessed via a form-capable browser (Mosaic, netscape, etc.), allow the user to perform input through text areas, menus, radio buttons, etc. This input is not ultimately handled by the browser. Instead, forms generally have a “submit” button. When this button is pressed, the input provided through the menus, text areas, etc. is sent by the browser to a “handler” program, which can be anywhere on the net. The sending browser then waits for a response from that program, which comes in the form of a new HTML document. See for example

```
http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/
↪fill-out-forms/overview.html
```

for details.

The following HTML terms provide a simple way of producing forms from a Prolog program:

`form(`*Addr*`)` Specifies the beginning of a form. *Addr* is the address (URL) of the program that will handle the form (translates to `<form method="POST" action="`*Addr*`">`).

`end_form` Specifies the end of a form (translates to `</form>`)

`checkbox(`*Name*`,`*State*`)` Specifies an input of type `checkbox` with name *Name*, *State*=on if the checkbox is initially checked (translates to an `<input>` element).

`radio(`*Name*`,`*Value*`,`*Selected*`)` Specifies an input of type `radio` with name *Name* (several radio buttons which are interlocked must share their name), *Value* is the the value returned by the button, if *Selected*=*Value* the button is initially checked (translates to an `<input>` element).

`input(`*Type*`,`*Atts*`)` Specifies an input of type *Type* with a list of attributes *Atts*. Possible values of *Type* are `text`, `password`, `submit`, `reset`, ... (translates to an `<input>` element).

`textarea(`*Name*`,`*Atts*`,`*Text*`)` Specifies an input text area of name *Name*. *Text* provides the default text to be shown in the area, *Atts* a list of attributes (translates to a `<textarea>` environment).

`menu(`*Name*`,`*Atts*`,`*Items*`)` Specifies a menu of name *Name*, list of attributes *Atts* and list of options *Items*. The elements of the list *Items* are marked with the prefix operator '`$`' to indicate that they are selected (translates to a `<select>` environment).

`form_reply` This is not HTML, rather, the HTTP protocol requires this content descriptor to be used by form handlers when replying (translates to "`Content-type:  text/html`").

#### 4.1.4 Extending the Package

Along with the obvious solution of changing the source code directly, the package has a built-in mechanism for extending it by adding clauses to the following predicate:

`html_expansion(`*Macro*`,`*Translation*`)` The new HTML structure *Macro* is equivalent to the standard one *Translation*. Care must be taken not to transform a structure into itself (this would lead to an infinite loop).

The package has these two equivalences predefined (just to make environments more LaTeXish):

```
html_expansion(bf(X),b(X)).
html_expansion(it(X),i(X)).
```

As an example, to define structures for the new HTML3 tabbing elements, one would define the following clauses:

```
html_expansion(settab(I),tabstop$[id(I)]).
html_expansion(tabto(I),tab$[to(I)]).
```

### 4.2 Predicates for Producing HTML Executables

An HTML executable is a standard executable which resides in a directory which is recognized by the corresponding `httpd` server as containing executables. The set of such directories normally includes `/cgi-bin`, but most `httpd` servers can be configured so that other directories are also included in this class. CGI stands for "Common Gateway Interface", see

`http://kuhttp.cc.ukans.edu/info/forms/forms-intro.html`

for a good introduction of CGI scripts and HTML forms.

When access to a standard document is requested by a browser, the `httpd` server normally simply returns the contents of the file (which is normally in HTML format). However, if the requested address corresponds to a file in an executable directory then the server first runs the executable and then returns to the requesting browser the output produced by such run (which is normally in HTML format or another format which can be handled by the browser). HTML requests to executables are generally information requests and Logic Programming is a natural choice for developing applications capable of fulfilling such requests.

Developing a WWW executable in Prolog simply amounts to producing a Prolog program which, when executed, returns an HTML document. The formatting facilities presented in the previous sections can be used for providing such a document. The Prolog executable can be produced using the usual techniques in Prolog, i.e., through a saved state or by generating a stand-alone executable. At CLIP we have also developed a simple package for creating "Prolog scripts", i.e., files containing source code which simply run when executed, in the same way as is usually done, for example, with shell scripts in UNIX (this package will be released soon).

The Prolog HTML package also contains some simple predicates which somewhat simplify the operation of producing a `cgi-bin` executable:

`save_cgi_bin(`*Name*`,`*EntryPoint*`)` creates an executable corresponding to the current state (using a saved state, only available in systems which support them). The name of the executable will be *Name* and the entry point when the executable is called will be *EntryPoint*. The executable will also contain an exception handler which produces suitable output (via an HTML document) if the application fails or gives run-time errors (see the next predicate). The warning icon used in these messages can be changed through the predicate `icon_address/2`. See the source file for details.

`html_protect(`*Goal*`)` Protects the execution of the goal *Goal* so that if it fails or gives run-time errors suitable HTML output is produced.

The following section explains the facilities provided by the package for creating form handlers, including the parsing of input from HTML forms.

## 4.3 Predicates for Creating Form Handlers

An HTML form handler is a special kind of HTML executable that is accessed when a form is submitted. I.e., it is the program designated in the form command. The predicates provided to simplify this task are:

`html_form_check` Used to ensure that the executable was invoked by an HTML form with a method of "`POST`".

`html_get_form_input(`*Dic*`)` Translates input from the form to a dictionary *Dic* of *attribute*=*value* pairs. It translates empty *value*s to `true`.

`html_get_value(`*Dic*`,`*Var*`,`*Val*`)` Gets value *Val* for attribute *Var* in dictionary *Dic*. Does not fail: value is `''` if not found.

`html_empty_value(`*V*`)` Useful to check that a value *V* from a text area is empty (can have spaces, newlines and linefeeds).

`default(`*Val*`,`*Default*`,`*NewVal*`)` Useful when the executable can be invoked either directly or by a form, to set form defaults. If *Val* is empty (because the executable was invoked directly) *NewVal*=*Default*, else *NewVal*=*Val*.

# 5 Example Files

We have created several example files to illustrate the use of the predicates provided by the package and to demonstrate its capabilities. For each example we list first the Prolog code (`.pl`) and then its output (`.html`).

`html_simple.pl` The code shown in section 4.1.

`html_simple.html` Shown in the same place.

`html_demo.pl` A file that illustrates the creation of HTML pages using the specific Prolog HTML structures defined in section 4.1.2.

```
:- ['/usr/local/lib/prolog/html.pl'].

main :-
  tell('html_demo.html'),
  output_html([
    start, nl,
    comment('This document was generated from Prolog'),
    title('Sample HTML document generated from Prolog'),
    heading(1,'Sample HTML document'),
    --,
    heading(2,'Miscellaneous'),
    'This is a ',ref('#label',it(reference)),' to ',nl,
    bf(['another ',it(point)]),' in this document.',$,
    'Let''s be ',nl,
    image('http://www.clip.dia.fi.upm.es/images/smile.happy.gif',[alt=':-)']),
```

```
    '!',
    preformatted(['These lines',
      ['are ',b(preformatted),'.'],
      'See?']),
    'We have left here a ',strong(free),' variable:',nl,X,$,
    'This is in verbatim: ',nl,
    samp(verbatim('<NOTE> write "&amp;" to insert an &')),\\,
    'But this is not:',
    samp('<NOTE> write "&amp;" to insert an &'),$,
    label(label,['This is the point referenced ',it('above.')]),
    heading(2,'Lists'),
    heading(3,tt(itemize)),
    itemize([one,two,['and ',3]]),
    heading(3,tt(enumerate)),
    enumerate([b(red),i(green),tt(blue)]),
    heading(3,tt(description)),
    description([
      (one,b(two),'Description of one and two'),
      (three,'Idem of three'),
      ['This ',b(also)]]),
    heading(3,'Nice itemize'),
    nice_itemize([pretty,bf(fancy),nice]),
    --,
    address('clip@dia.fi.upm.es'),
    end
  ]),
  told.
```

`html_demo.html` Output from the above code.

```
<html>
<!-- This document was generated from Prolog -->
<title>Sample HTML document generated from Prolog</title>
<h1>Sample HTML document</h1>

<hr>
<h2>Miscellaneous</h2>
This is a <a href="#label"><i>reference</i></a> to
<b>another <i>point</i></b> in this document.
<p>Let's be
<img src="http://www.clip.dia.fi.upm.es/images/smile.happy.gif" alt=":-)">!<pre>
These lines
are <b>preformatted</b>.
See?
</pre>We have left here a <strong>free</strong> variable:
```

```
<b>**Warning free variable**</b>
<p>This is in verbatim:
<samp>&lt;NOTE&gt; write &quot;&amp;amp;&quot; to insert an &amp;</samp><br>
But this is not:<samp><NOTE> write "&amp;" to insert an &</samp>
<p><a name="label">This is the point referenced <i>above.</i></a><h2>Lists</h2>
<h3><tt>itemize</tt></h3>
<ul>
<li> one
<li> two
<li> and 3
</ul><h3><tt>enumerate</tt></h3>
<ol>
<li> <b>red</b>
<li> <i>green</i>
<li> <tt>blue</tt>
</ol><h3><tt>description</tt></h3>
<dl>
<dt>one
<dt><b>two</b>
<dd>Description of one and two
<dt>three
<dd>Idem of three
<dd>This <b>also</b>
</dl><h3>Nice itemize</h3>
<dl>
<dd><img src="http://www.clip.dia.fi.upm.es/images/redball.gif" align="bottom"
 alt="*">pretty
<dd><img src="http://www.clip.dia.fi.upm.es/images/redball.gif" align="bottom"
 alt="*"><b>fancy</b>
<dd><img src="http://www.clip.dia.fi.upm.es/images/redball.gif" align="bottom"
 alt="*">nice
</dl>
<hr>
<address>clip@dia.fi.upm.es</address></html>
```

`html_simple_form.pl` A simple file which illustrates the creation of HTML form handlers using the Prolog HTML structures for forms defined in section 4.1.3. It also uses the predicates for producing HTML executables (explained in section 4.2) and for creating form handlers (explained in section 4.3).

```
:- ['/usr/local/lib/prolog/html.pl'].

main :-
  html_get_form_input(Info),
```

```
    html_get_value(Info,person_name,Name),
    response(Name,Response),
    output_html([
      form_reply,
      start,
      image('http://www.clip.dia.fi.upm.es/images/clip.gif'),nl,
      heading(2,'Toy Address Server'),
      --,
      Response,
      form('http://www.clip.dia.fi.upm.es/cgi-bin/html_simple_form'),
      'Click here, enter name, and press return:',
      \\,
      input(text,[name=person_name,size=20]),
      end_form,
      end]).

response(Name, []) :-
        html_empty_value(Name), !.
response(Name, ['Available info on ', bf(Name), ':', Info]) :-
        info(Name,Info), !.
response(Name, ['No available info on ', bf(Name), '.']).

info(clip,   '+341-336-7448').
info(manuel, '+341-336-7435').
info(daniel, '+341-336-7448').

:- save_cgi_bin(html_simple_form,main).
```

`html_simple_form.html` Output the first time the form handler is invoked.

```
Content-type: text/html

<html><img src="http://www.clip.dia.fi.upm.es/images/clip.gif">
<h2>Toy Address Server</h2>

<hr>
<form method="POST" action="http://www.clip.dia.fi.upm.es/cgi-bin/html_simple_form">
Click here, enter name, and press return:<br>
<input type="text" name="person_name" size="20"></form></html>
```

`html_simple_form` The executable produced by `html_simple_form.pl`. You can try it out at `http://www.clip.dia.fi.upm.es/cgi-bin/html_simple_form`.

`html_forms.pl` A more involved example of form handling illustrating more uses of the Prolog HTML structures for forms defined in section 4.1.3.

```
:- ['/usr/local/lib/prolog/html.pl'].

main :-
  html_get_form_input(Info),
  html_get_value(Info,mood,Mood0),
  html_get_value(Info,potatoes,Potatoes),
  html_get_value(Info,pizza,Pizza),
  html_get_value(Info,coke,Coke),
  html_get_value(Info,pass,Password),
  html_get_value(Info,text,Text),
  html_get_value(Info,menu1,Number),
  findall(Color,member(menu2=Color,Info),Colors),
  compute_reply(Mood0,Potatoes,Pizza,Coke,Password,Text,Number,Colors,Reply),
  default(Mood0,happy,Mood),
  output_html([
    form_reply,
    start,
    title('Sample Form generated from Prolog'),
    heading(1,'Sample HTML form'),
    Reply,
    --,
    heading(2,'This is a form:'),
    form('http://www.clip.dia.fi.upm.es/cgi-bin/html_forms'),
    'Please select mood:',\\,
    radio(mood,happy,Mood),nl,
    image('http://www.clip.dia.fi.upm.es/images/smile.happy.gif',
      [align=middle]),
    nl,
    radio(mood,sad,Mood),nl,
    image('http://www.clip.dia.fi.upm.es/images/smile.sad.gif',
      [align=middle]),
    $,
    'What ',strong(do),' you want?',\\,
    checkbox(potatoes,Potatoes), 'Potato(e)s',\\,
    checkbox(pizza,Pizza), 'Pizza',\\,
    checkbox(coke,Coke), 'Coke',
    --,
    'Please write a password:',nl,
    input(password,[name=pass,size=9,maxlength=8]),$,
    textarea(text,[rows=5,cols=20],'Write here something'),$,
    'You can choose here:', menu(menu1,[],[one,$two,three]),$,
    'Also here:', menu(menu2,[multiple],[$red,$green,blue]),$,
    input(submit,[value='Send values']),nl,
    input(reset,[value='Reset values']),
    end_form,
```

```
    --,
    address('clip@dia.fi.upm.es'),
    end]).

compute_reply('',_,_,_,_,_,_,_,'') :- !.
compute_reply(Mood,Potatoes,Pizza,Coke,Passwd0,Text0,Number,Colors0,Reply) :-
  compute_food(Potatoes,Pizza,Coke,Food),
  compute_text(Text0,Text),
  compute_password(Passwd0,Passwd),
  compute_colors(Colors0,Colors),
  Reply = [
    --,
    heading(2,'Submitted data:'),
    'You are ',strong(Mood),'.',\\,
    'You want ',strong(Food),\\,
    'The password is ',tt(verbatim(Passwd)),'.',\\,
    'In the text area you wrote: ',tt(verbatim(Text)),$,
    'You chose number ',tt(Number),' and color(s):',
    preformatted(Colors)].

compute_food('','','','no food.').
compute_food(on,'','','potatoes.').
compute_food('',on,'','pizza.').
compute_food('','',on,'coke.').
compute_food(on,on,'','potatoes and pizza.').
compute_food('',on,on,'pizza with coke.').
compute_food(on,'',on,'potatoes with coke.').
compute_food(on,on,on,'potatoes and pizza with coke.').

compute_text(T0,T) :-
  html_empty_value(T0) -> T = '{Nothing}' ; T = T0.

compute_password(P0,P) :-
  P0 = true -> P = '{none}'; P = P0.

compute_colors(C0,C) :-
  C0 = [] -> C = [none] ; C = C0.

member(X,[X|_]).
member(X,[_|Xs]) :- member(X,Xs).

:- save_cgi_bin(html_forms,main).
```

`html_forms.html` Output the first time the form handler is invoked.

```
Content-type: text/html
```

```
<html><title>Sample Form generated from Prolog</title>
<h1>Sample HTML form</h1>

<hr>
<h2>This is a form:</h2>
<form method="POST" action="http://www.clip.dia.fi.upm.es/cgi-bin/html_forms">
Please select mood:<br>
<input name="mood" type="radio" value="happy" checked>
<img src="http://www.clip.dia.fi.upm.es/images/smile.happy.gif" align="middle">
<input name="mood" type="radio" value="sad">
<img src="http://www.clip.dia.fi.upm.es/images/smile.sad.gif" align="middle">
<p>What <strong>do</strong> you want?<br>
<input name="potatoes" type="checkbox">Potato(e)s<br>
<input name="pizza" type="checkbox">Pizza<br>
<input name="coke" type="checkbox">Coke
<hr>
Please write a password:
<input type="password" name="pass" size="9" maxlength="8">
<p><textarea name="text" rows="5" cols="20">Write here something</textarea>
<p>You can choose here:<select name="menu1"><option>one</option>
<option selected>two</option>
<option>three</option>
</select>
<p>Also here:<select name="menu2" multiple><option selected>red</option>
<option selected>green</option>
<option>blue</option>
</select>
<p><input type="submit" value="Send values">
<input type="reset" value="Reset values"></form>
<hr>
<address>clip@dia.fi.upm.es</address></html>
```

`html_forms` The executable produced by `html_forms.pl`. You can try it out at `http://www.clip.dia.fi.upm.es/cgi-bin/html_forms`.

Additionally, you can try out `webchat`, a geographical database server with natural language interface (good old Chat) starting from `http://www.clip.dia.fi.upm.es/miscdocs/webchat_info.html`.

# 6 Coming soon

The HTML package, as it is right now, allows producing documents and developing information servers. However, it would also be useful to be able to directly access the net from a Prolog application, and through a translation from HTML to Prolog terms, read remote pages and manipulate their contents, for example to perform an intelligent keyword analysis of the text within or to find pointers within the document and in turn follow them. We are working on adding the following features to support this task:

- Support for the HTTP protocol through builtins.
- A reversible grammar that can also parse HTML documents and turn them into Prolog terms.
- Integration with a "Concurrency/Distribution" package.

The next release will (hopefully) include these features. We are also working on a release of our Prolog "Scripts" and Concurency/Distribution packages, which allow writing executable LP/CLP scripts and concurrent execution, access to remote code, active objects, etc [1], with standard LP/CLP systems.

Please send any comments, bugs, suggestions, to `clip@dia.fi.upm.es`.

# 7 Acknowledgments

This package has been written by Manuel Hermenegildo and Daniel Cabeza, using input from L. Naish's forms and F. Bueno's previous Chat interface. Thanks are due to Pedro López and other members of the CLIP group for useful feedback on this document and on the package itself.

# References

[1] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proccedings of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. Available online at ftp://clip.dia.fi.upm.es/pub/papers/ciao-dis-impl-parimp.ps.Z.

[2] M. Carlsson. *Sicstus Prolog User's Manual*. P.O. Box 1263, S-16313 Spanga, Sweden, February 1988. Available online at http://www.sics.se/ps/sicstus/sicstus_toc.html.

[3] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proccedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995. Available online at ftp://clip.dia.fi.upm.es/pub/papers/ciao-report-ilps95ws_final.ps.Z.

[4] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994. Available online at ftp://clip.dia.fi.upm.es/pub/papers/ciao_ppcp.ps.Z.

[5] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991. Available online at ftp://clip.dia.fi.upm.es/pub/papers/imp_ngc_final.ps.Z.

[6] L. Lamport. *LATEX User's Guide & Reference Manual*. Addison-Wesley Publishing Company, Inc., 1986. Online information on TeX and LaTeX is available at http://curia.ucc.ie/info/TeX/menu.html.