Computational Logic

WWW Programming Using LP/CLP Systems

LP/CLP, the Internet, and the WWW

- Logic and Constraint Logic Programming can be an attractive alternative for Internet/WWW programming.
- Shared with other net programming tools:
  - dynamic memory management,
  - well-behaved structure (and pointer!) manipulation,
  - robustness, compilation to architecture-independent bytecode, ...
- In addition:
  - powerful symbolic processing capabilities,
  - dynamic databases,
  - search facilities,
  - grammars,
  - sophisticated meta-programming / higher order,
  - easy code (agent) motion,
  - well understood semantics, ...

LP/CLP, the Internet, and the WWW

- Most public-domain and commercial LP/CLP systems:
  - ◇ either already have Internet connection capabilities (e.g., socket interfaces),
  - ◇ or it is relatively easy to add it to them (e.g., through the C interface)

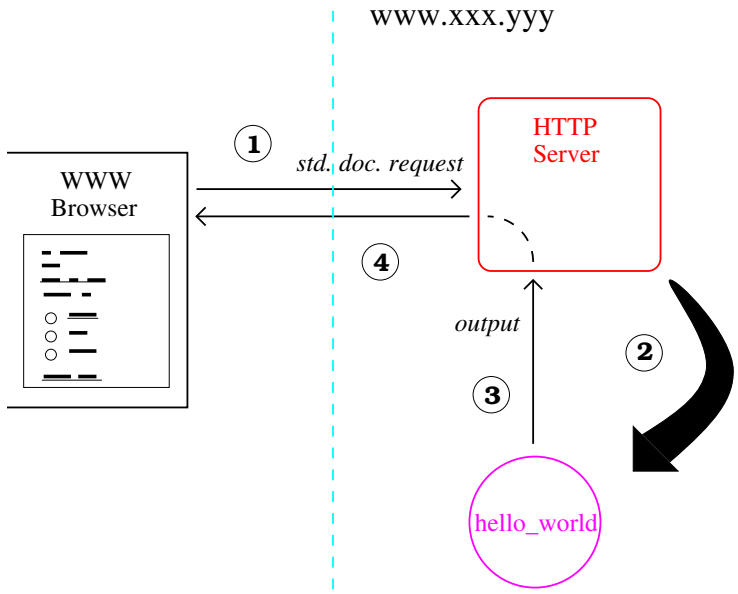
  (e.g., Quintus, LPA, PDC, Amzi!, IF-Prolog, Eclipse, SICStus, BinProlog, SWI, PrologIV, CHIP, Ciao, etc.)
- Some additional "glue" needed to make things really convenient:
  - ◇ We present several techniques for "filling in these gaps" (many implemented as public domain libraries).
  - ◇ Some commercial systems also include packages that provide similar high-level functionality.
- In doing this we also work towards answering the question:
  - ◇ What are useful characteristics of particular LP/CLP systems in this context?

Global Outline

- PART I: *WWW programming*
  - Writing cgi-scripts.
  - Seeing HTML structured documents as Herbrand terms.
  - Producing HTML *forms*.
  - Writing form handlers.
  - HTML templates.
  - Accessing and parsing WWW documents.
  - Accessing code posted at HTTP addresses.
  - XML, VRML, etc.
- PART II: *Distributed/agent programming*

Writing Basic CGI-bin Applications

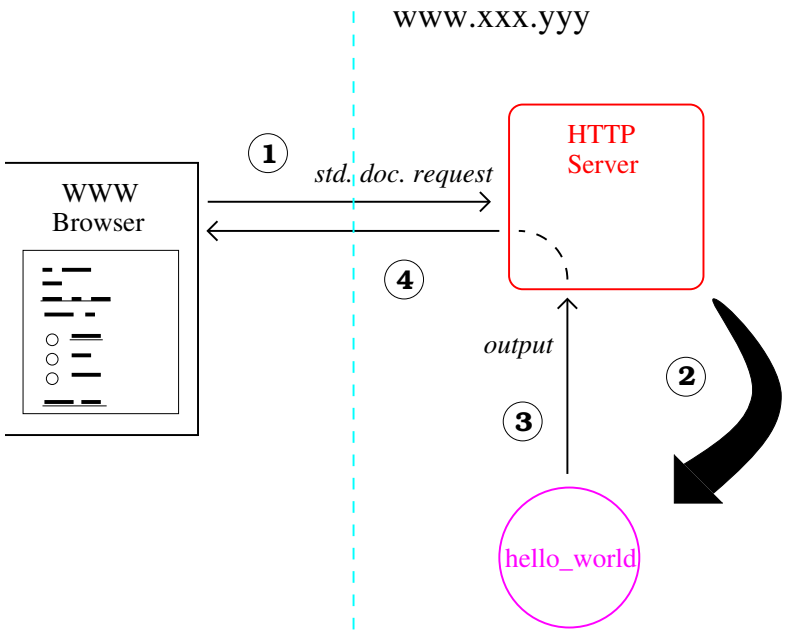
1. A standard URL is selected in a browser (client), which is the address of the CGI application, e.g.: `http://www.xxx.yyy/cgi_bin/hello_world`
2. The browser sends it to the corresponding HTTP server.
3. The executable "`hello_world`" (in directory `cgi_bin`) is started by HTTP server.
4. Executable output (stdout) (which has to be in HTML –or MIME– format) is taken by the HTTP server, and passed on to the client Browser, which displays it.



An example: UNIX csh

See `http://www.clip.dia.fi.upm.es/demo/pillow/hw_csh.cgi`

```
#!/bin/tcsh

echo "Content-type: text/html"
echo ""
echo "<HTML>"
echo "Hello <B>world</B>."
echo "</HTML>"
echo ""
```



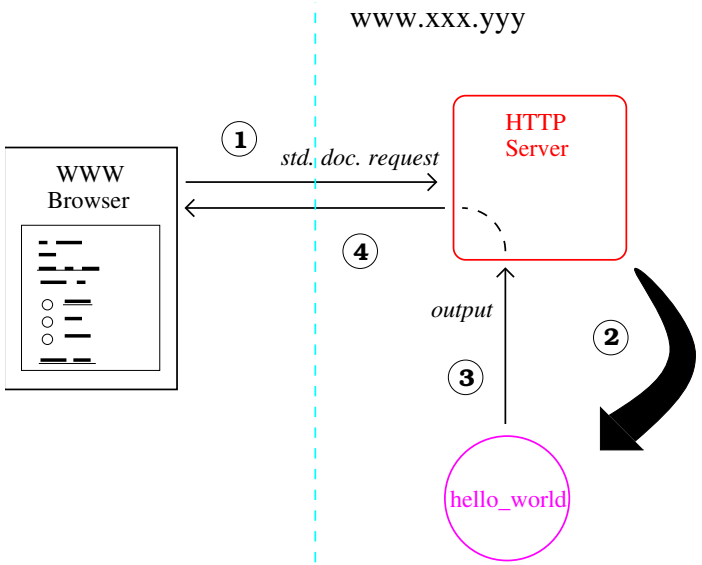
- Similarly, with DOS/Win `.bat` files, etc.
- The CGI application often has to be:
  - in a special directory (e.g., `/usr/local/etc/httpd/cgi-bin`),
  - or it must have a ".cgi" ending.

Writing Basic CGI-bin Applications in LP/CLP

See `http://www.clip.dia.fi.upm.es/demo/pillow/hw_prolog.cgi`

- A first approach:

```
main(_) :-
  write('Content-type: text/html'),
  nl, nl,
  write('<HTML>'),
  write('Hello <B>world</B>.'),
  write('</HTML>').
```



- And the executable can be generated, e.g., by:

```
ciaoc -o hw_prolog.cgi hw_prolog
```

Scripting Languages

- "Scripting" languages (perl, csh, ...) popular for writing CGI apps:
  - CGI's: often non-numerical, small- to medium-sized apps.
  - Strong support for symbol manipulation.
  - No compilation necessary.
  - The network is slow anyway.
  - Small "executable" size (source file!).
- A role for LP/CLP?
  - LP/CLP languages can be great as scripting languages: built-in grammars, databases, interpreter available, fast compilation, ...
  - But some shortcomings: awkward executable creation, large executables, ...

Effective LP/CLP scripts

See `http://www.clip.dia.fi.upm.es/demo/pillow/hw_pshell.cgi`

- LP/CLP systems can easily be used as scripting languages (e.g., for unix):

```
#!/bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

main(_) :-
  write('Content-type: text/html'), nl, nl,
  write('<HTML>'),
  write('Hello <B>world</B>.'),
  write('</HTML>').
```

  where `ciao_shell` is an *executable* which:

  - skips the first line(s),
  - loads (consults or compiles) the rest of the file, and
  - starts at `main/1`.

Effective LP/CLP scripts (Contd.)

- Can easily be made to “cache” compilations using bytecode files.
- Available in Ciao Prolog distribution.
- Can also be done in several other ways (`.sh` files, `.bat` files, etc.).
- Above solution also available for SICStus from `ftp:clip.dia.fi.upm.es`
- (very useful also for writing “filters” –e.g., for unix pipes–, etc.)

Relating HTML code and Prolog Terms

- HTML is structured: it is possible to reflect this structure as Prolog terms.
- Allows viewing any WWW page as a Herbrand term and manipulating it easily.
- Ideally, provide bidirectional conversion between a string representing the HTML code and its term representation.
- This can be easily done, for example with DCGs.
- E.g, predicates for this purpose provided in PiLLoW:
  - `html2terms(`$ASCII$`,`$Terms$`)` (and `xml2terms(`$ASCII$`,`$Terms$`)`)
    Relates a list of HTML terms and a list of ASCII characters (reversible).
  - `output_html(`F`)`
    Sends to the standard output the text corresponding to the HTML term F (calls `html2terms/2` and then makes the necessary calls to `write/1`).

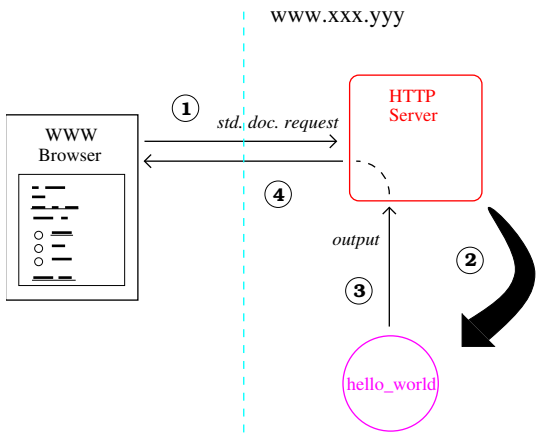
(PiLLoW: public domain WWW/LP interface library –a Ciao library, but versions available for several popular LP/CLP systems)

Relating HTML code and Prolog Terms

See `http://www.clip.dia.fi.upm.es/demo/pillow/hw_pillow.cgi`

- Example:

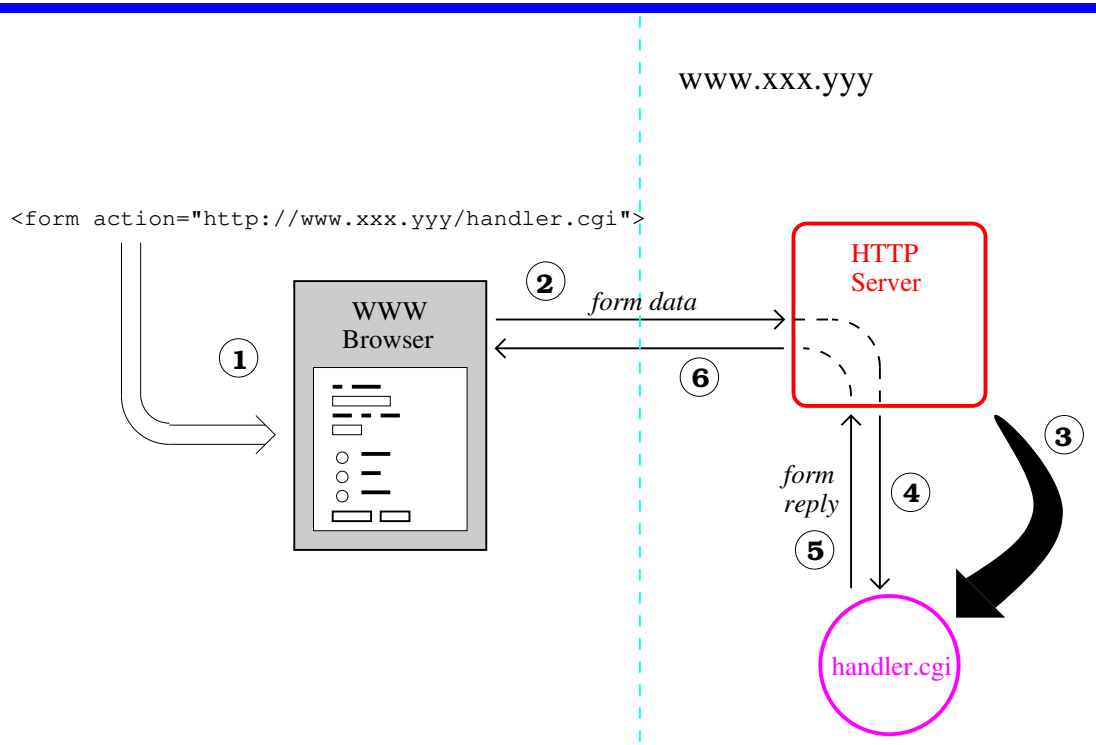
```
#! /bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).

main(_) :-
    T = [ 'Content-type: text/html',
          html( [ 'Hello',
                  b(world)
                ] )
        ]),
    output_html(T).
```



Relating HTML code and Prolog Terms

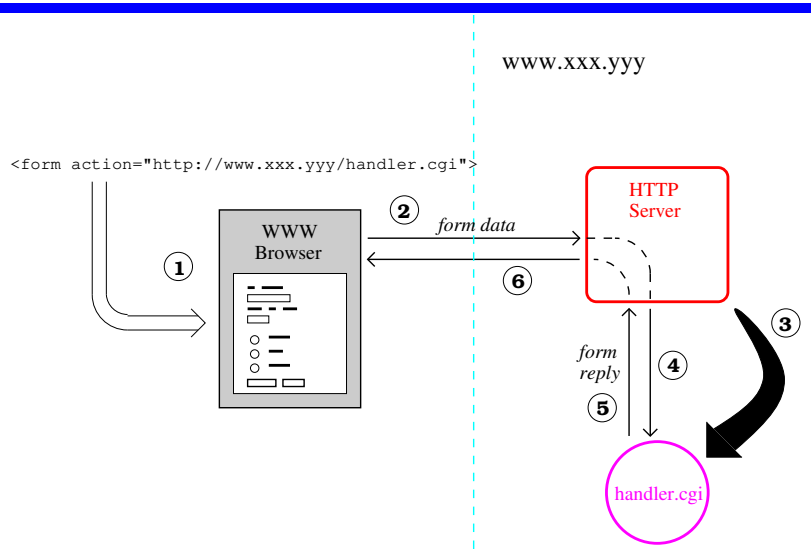
- PiLLoW general HTML structures (can represent any HTML code):
  - *Name* `$` *Atts* ('`$/2`' is defined as an infix, binary operator.)
    `img$[src='images/map.gif',alt='A map',ismap]` ⇒
    `<img src="images/map.gif" alt="A map" ismap>`
  - *name*(*Text*) (Term with functor *name*/1)
    `address('clip@dia.fi.upm.es')` ⇒
    `<address>clip@dia.fi.upm.es</address>`
  - *name*(*Atts*, *Text*) (Term with functor *name*/2)
    `a([href='http://www.xx.y/'],'XX home')` ⇒
    `<a href="http://www.xx.y/">XX home</a>`
  - env(*Name*, *Atts*, *Text*)
    `env(a,[href='http://www.xx.y/'],'XX home')` ⇒
    `<a href="http://www.xx.y/">XX home</a>`
- Also, specific structures to simplify HTML creation.

Responding to Input: Forms



- A form is a standard HTML document (supported by all browsers) with
  - ◇ "fields" which can be filled in (each has a *name*),
  - ◇ a "submit" button,
  - ◇ URL of CGI application that will handle the input (the "handler")

Responding to Input: Forms



- Operation (when hitting the “submit” button):
  1. We assume the URL of the handler is `http://www.xxx.yyy/handler.cgi`.
  2. Handler URL and form data (input) are passed to server.
  3. Server starts handler and passes form data (via stdin or standard file name) associating field names to entered values.
  4. The handler produces the appropriate reply,
  5. which is passed back to the browser.

Writing Form Handlers in LP/CLP

- Use same techniques as with standard CGI apps.
- Only complication is form data parsing (names/values).
- Good solution: implement a parser (easy in LP/CLP) and produce an attribute-value pair list or dictionary.
- Enables the symbolic treatment of form data, hiding the low-level protocol behind.

E.g., predicates provided in PiLLoW:

- `get_form_input(`Dic`)` Translates input from the form to a dictionary Dic of $attribute$=$value$ pairs. This is implemented using a simple DCG parser.
  `get_form_input(Dict)` $\Rightarrow$ `Dict = [name='Anna', age=23]`
- `get_form_value(`Dic`,`Var`,`Val`)` Gets value Val for attribute Var in dictionary Dic.
- `form_empty_value(`V`)` Useful to check that a value V from a text area is empty.
- `my_url(`URL`)` Returns the Uniform Resource Locator (WWW address) of form.

A browser can be used as a graphical interface!

Writing Form Handlers in LP/CLP: Example

See `http://www.clip.dia.fi.upm.es/demo/pillow/simple_form.html`

- A simple form:

```
<html>
<hr>
<h2>Please enter input (person_name):</h2>
<form  method="POST"
       action="http://localhost/~clip/demo/pillow/simple_handler.cgi">
<input type="text" name="person_name" size="40">
<input type="submit" value="Submit">
</form>
<hr>
</html>
```

Writing Form Handlers in LP/CLP: Example

See `http://www.clip.dia.fi.upm.es/demo/pillow/simple_handler.cgi`

- A simple form handler (`simple_handler.cgi`):

```
#!/bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    Answer = [  hr$[],
                h2('You submitted the name: '),
                em(Name),
                hr$[] ],
    output_html([ 'Content-type: text/html', html(Answer) ]).
```

Producing Forms from Programs: Example

See `http://www.clip.dia.fi.upm.es/demo/pillow/simple_form_pillow.cgi`

- The form itself can be the result of running a program:

```
#!/bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).
main(_) :-
    Form = [
      hr$[],
      h2('Please enter input (person_name):'),
      form([ method=post,
             action='http://localhost/~clip/demo/pillow/simple_handler.c
           [ input$[type=text,name=person_name,size=40],
             input$[type=submit,value='Submit'] ] ),
      hr$[] ],
    output_html([ 'Content-type: text/html', html(Form) ]).
```

Producing Forms from Programs: Example

See
`http://www.clip.dia.fi.upm.es/demo/pillow/simple_form_pillow_sugar.cgi`

- Or using some minor syntactic sugar (really, deprecated):

```
#!/bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).
main(_) :-
    Form = [  --,
      h2('Please enter input (person_name):'),
      start_form('http://localhost/~clip/demo/pillow/simple_handler.cgi'
      input(text,[name=person_name,size=40]),
      input(submit,[value='Submit']),
      end_form,
              -- ],
    output_html([ cgi_reply, html(Form) ]).
```

Combining the Form Producer and Handler

See `http://www.clip.dia.fi.upm.es/demo/pillow/combined_form.cgi`

```
#!/bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).
main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    output_html([ cgi_reply, html( [
              --, h2('You submitted the name: '), em(Name),
              --, h2('Please enter input (person_name):'),
              start_form, %% Refers to self!
              input(text,[name=person_name,size=40]),
              input(submit,[value='Submit']),
              end_form,
              -- ] ) ] ).
```

A Phones Database

See `http://www.clip.dia.fi.upm.es/demo/pillow/phone_db.cgi`

```
response(Name, []) :- form_empty_value(Name), !.
response(Name, ['Phone number for ',bf(Name),' is ',Info, --]) :-
        phone(Name,Info), !.
response(Name, ['No phone number available for ',bf(Name), '.', --]).

%% Database
phone('CLIP',   '336-7448').
phone('Paco',   '554-5225').
phone('Daniel', '460-0569').
```

A Phones Database (Contd.)

```
main(_) :-
        get_form_input(Input),
        get_form_value(Input,person_name,Name),
        response(Name,Response),
        output_html([
          cgi_reply,
          begin(html),
          title('Simple CLIP telephone database'),
          begin(body,[background='/demo/images/Clip_bg.gif']),
          center( [
            image('/demo/images/clip.gif'),
            heading(2,'Simple CLIP telephone database'),
            --, Response,
            start_form,
            'Click here, enter name of clip member,
             and press return:',  \\,
            input(text,[name=person_name,size=20]), --,
            end_form,
            image('/demo/images/pillow_d.gif') ]),
          end(body), end(html)]).
```

HTML/XML Templates

- In the previous examples layout is hard-coded.
- Sometimes desirable to have layout be an input.
- One solution is to use *templates*:
  - ◇ File with standard HTML code,
  - ◇ containing “slots”,
  - ◇ which are given an identifier by means of a special tag.
- Support predicates in PiLLoW:
  - ◇ `html_template(`$Chars$`,` $Terms$`,` $Dict$`)`
    - * $Chars$ is the HTML/XML code with the slots.
    - * $Terms$ is the PiLLoW term with variables (*holes*) in place of the slots.
    - * $Dict$ is a list of $name$=$Variable$ pairs relating the *holes* and the slot identifiers.
- The template can be created with a standard WYSIWYG HTML editor.

Example of template for phones db

See http://www.clip.dia.fi.upm.es/demo/pillow/html_template.html

```
<HTML><HEAD><TITLE>Simple CLIP telephone database</TITLE></HEAD>
<BODY background="/demo/images/Clip_bg.gif">
<CENTER>
<IMG src="/demo/images/clip.gif">
<H2>Simple CLIP telephone database</H2>
<HR>
<V>response</V>
<FORM method="POST">
Click here, enter name of clip member, and press return:<BR>
<INPUT type="text" name="person_name" size="20">
<HR>
</FORM>
<IMG src="/demo/images/pillow_d.gif">
</CENTER></BODY></HTML>
```

Phones db with template

See http://www.clip.dia.fi.upm.es/demo/pillow/phone_db_template.cgi

```
#! /bin/sh
exec /home/clip/bin/ciao-shell $0 $* # -*- mode: ciao; -*-

:- include(library(pillow)).
:- use_module(library(file_utils),[file_to_string/2]).

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    file_to_string('html_template.html', Contents),
    html_template(Contents, HTML_terms, [response = Response]),
    output_html([cgi_reply|HTML_terms]).

% response/2 and phone/2 as before.
```

Accessing WWW/Internet documents from LP/CLP

- The HTTP, FTP, etc. protocols are ASCII protocols which can be added relatively easily to an LP/CLP system (provided it has a socket interface or equivalent).
- Applications: search tools, content analyzers, reading html templates, etc. Also access to remote modules via WWW:

```
:- use_module('http://www.xx.y/prolog/p.pl')
```

- E.g., P*i*LL*o*W protocol support:
  - `fetch_url(`*URL*`,` *Request*`,` *Response*`)` –
  - Some *Request* options:
    - `head`: only interested in the header.
    - `timeout(`*Time*`)`: specifies number of seconds for timeout (fails).
    - `if_modified_since(`*Date*`)`
    - `authorization(`*Scheme*`,`*Param*`)`

Accessing WWW/Internet documents from LP/CLP

- (P*i*LL*o*W protocol support example Contd.):
  - Some possible elements of *Response*:
    - `content(`*Content*`)`: document as a list of characters.
    - `status(`*Type*`,`*Code*`,`*Phrase*`).`
    - `last_modified(`*Date*`).`
    - `expires(`*Date*`).`
    - `location(`*URL*`)` (document has moved).
  - `url_info(`*URL*`,` *Info*`)`: parses a URL.
- Example:

```
?- url_info('http://www.xx/foo.html',UI), fetch_url(UI,[],R),
member(content(C),R), html2terms(C,Terms).
```

Accessing WWW/Internet Example: Link Checker

See `http://www.clip.dia.fi.upm.es/demo/pillow/check_links.cgi`

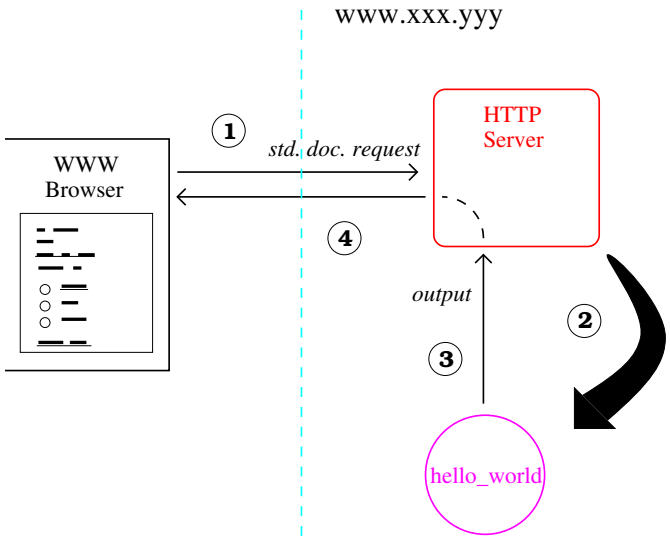
```
check_links(URL,BadLinks) :-
        url_info(URL,URLInfo),
        fetch_url(URLInfo,[],Response),
        member(content_type(text,html,_),Response),
        member(content(Content),Response),
        html2terms(Content,Terms),
        check_source_links(Terms,URLInfo,[],BadLinks).

check_source_links([],_,BL,BL).
check_source_links([E|Es],BaseURL,BL0,BL) :-
        check_source_links1(E,BaseURL,BL0,BL1),
        check_source_links(Es,BaseURL,BL1,BL).
check_source_links1(env(a,AnchorAtts,_),BaseURL,BL0,BL) :-
        member((href=URL),AnchorAtts), !,
        check_link(URL,BaseURL,BL0,BL).
check_source_links1(env(_Name,_Atts,Env_html),BaseURL,BL0,BL) :- !,
        check_source_links(Env_html,BaseURL,BL0,BL).
check_source_links1(_,_,BL,BL).
```

Accessing WWW/Internet Example: Link Checker (Contd.)

```
check_link(URL,BaseURL,BL0,BL) :-
        url_info_relative(URL,BaseURL,URLInfo), !,
        fetch_url_status(URLInfo,Status,Phrase),
        ( Status \== success ->
          name(P,Phrase),
          name(U,URL),
          BL = [badlink(U,P)|BL0]
        ; BL = BL0
        ).
check_link(_,_,BL,BL).

fetch_url_status(URL,Status,Phrase) :-
        fetch_url(URL,[head,timeout(20)],Response), !,
        member(status(Status,_,Phrase),Response).
fetch_url_status(_,timeout,timeout).
```

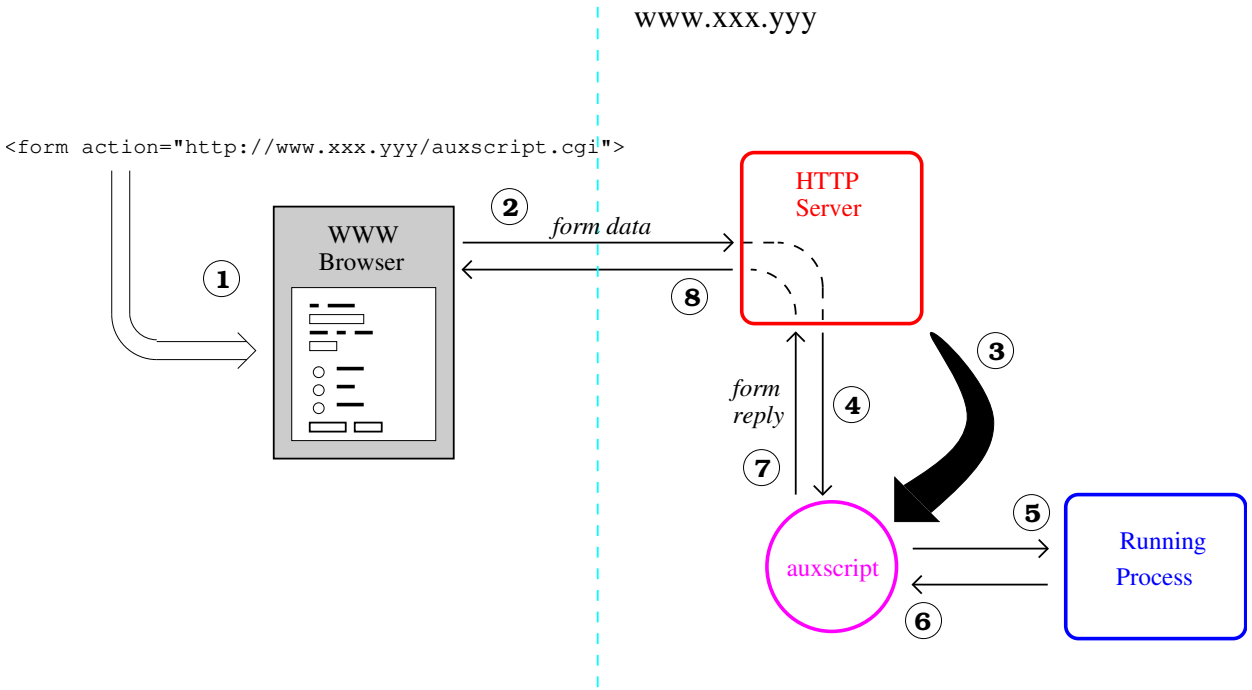
Limitations of CGI

- The cgi-bin interface dictates that a handler of a form starts and terminates for each interaction.
- Thus, form handlers in principle do not have state.
- State can in fact be passed through the form interface (*using info in hidden fields*).
- However, for a large application, starting and stopping can be very inefficient.



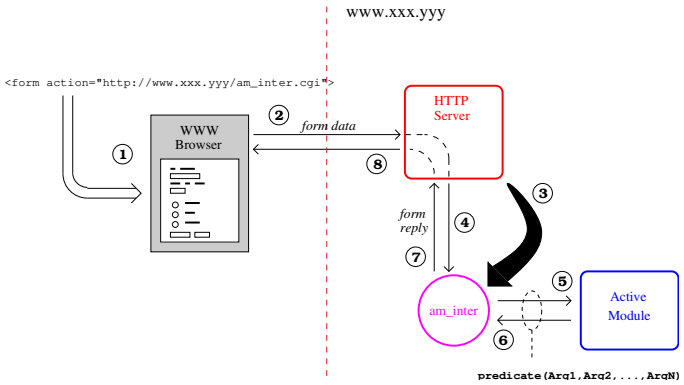
Solving the Limitations of CGI

- Standard solution: make the application be a permanently running process. A small CGI script (often written in perl) connects and disconnects from it for every interaction.



Solving the Limitations of CGI

- In LP/CLP:
  - The running process is a standard application.
  - The CGI executable can be an LP/CLP script or a C/perl/... program (e.g., ALS includes PiLLoW + such a solution)
  - Communication by standard means: sockets, blackboards, etc.
  - Several solutions proposed for dealing with several running sessions at the same time (in essence, concurrency is needed).
- "**Active modules**" (active objects) can be used well for this purpose.



Active Modules / Active Objects

- Modules to which computational resources are attached.
- High-level model of client-server interaction.
- An active module is a network-wide server for the predicates it exports.
- Any module or application can be converted into an “active module” (active object) by compiling it in a special way (creates an executable with a top-level listener).
- Procedures can be imported from remote “active modules” via a simple declaration: E.g. `:- use_active_module(Name, [`P_1/N_1`, `P_2/N_2`,...]).`
- Calls to such imported procedures are executed remotely in a transparent way.
- Typical application: client-server. Client imports module which exports the functionality provided by server. Access is transparent from then on.
- Can be built as an abstraction on top of ports/sockets
  (see our free library for CIAO, SICStus and other systems).

Using Active Modules: An Example

- Server code (active module), file `database.pl`:

```
:- module(database, [stock/2]).

stock(p1, 23).
stock(p2, 45).
stock(p3, 12).
```

- Compilation: "`ciaoc -a` *address publishing method* `database`" or:

```
?- make_actmod('/home/clip/public_html/demo/pillow/database.pl',
               'actmods/filebased_publish').
```

produces executable called `database`.

- Active module started as a process – e.g., in unix:
  `database &`

Using Active Modules: An Example

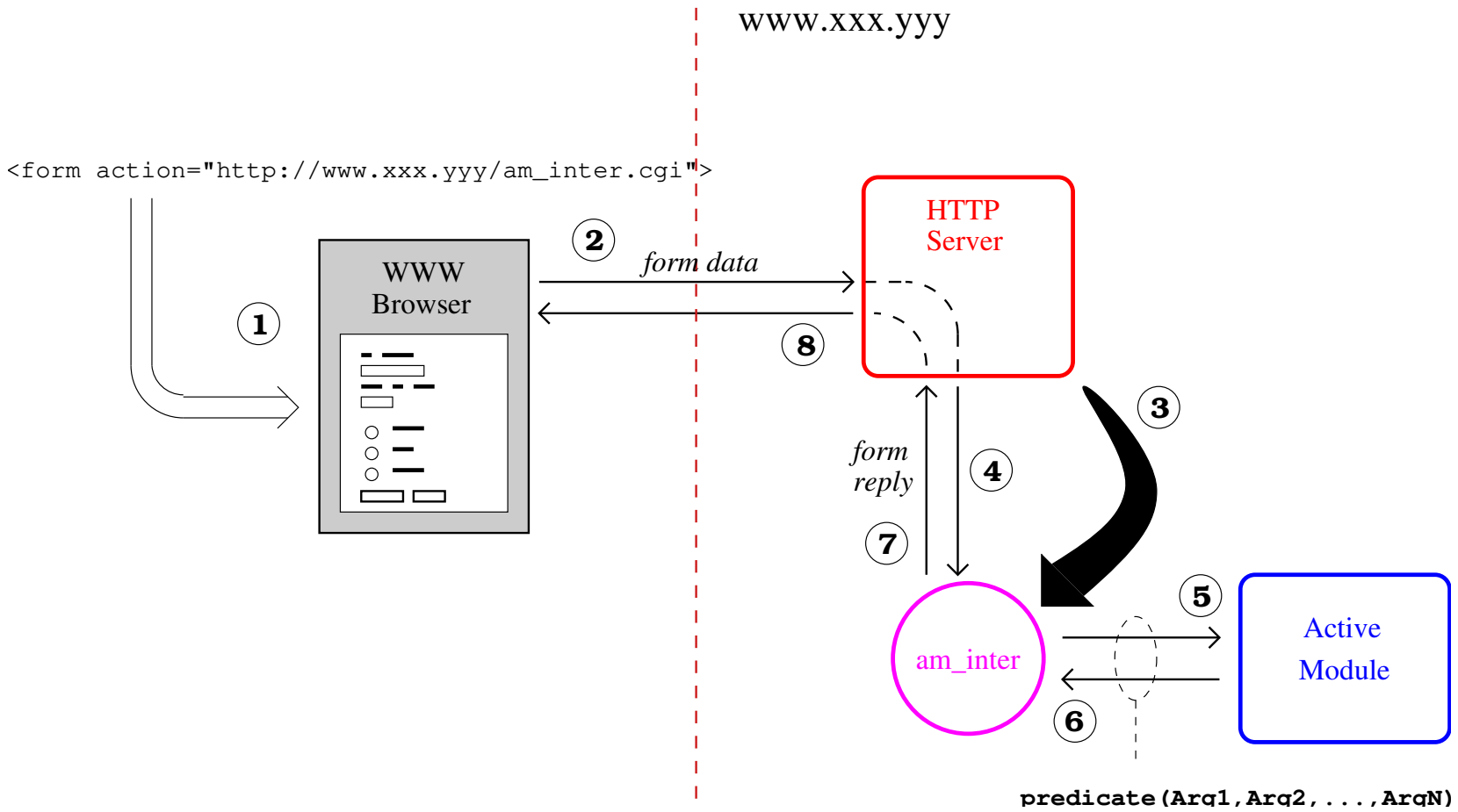
- Client (file `sales.pl`):

```
:- module(sales,[need_to_order/1],[actmods]).
:- use_active_module(database, [stock/2]).
:- use_module(library('actmods/filebased_locate')).

need_to_order(P) :-
        stock(P, S),
        S < 20.
```

- Usage:

```
?- use_module(sales).
?- need_to_order(X).
```

Application: Active Modules as Form Servers



Phone DB Using Active Modules: Server

- Server (the active module):

```
:- module(_,[process_form/2],[pillow]).
:- use_module(library(file_utils),[file_to_string/2]).

process_form(Input,Output) :-
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    file_to_string('html_template.html', Contents),
    html_template(Contents, HTML_terms, [response = Response]),
    Output = [cgi_reply|HTML_terms].

response(Name, []) :- form_empty_value(Name), !.
response(Name, ['Phone number for ',bf(Name),' is ',Info, --]) :-
        phone(Name,Info), !.
response(Name, ['No phone number available for ',bf(Name), '.', --]).

%% Database
phone('CLIP',   '336-7448').
phone('Paco',   '554-5225').
phone('Daniel', '460-0569').
```

Phone DB Using Active Modules: Client

See `http://www.clip.dia.fi.upm.es/demo/pillow/phone_db_client.cgi`

- Client (the `.cgi` using the active module):

```
:- module(_,[main/1],[actmods,pillow]).

:- use_active_module(phone_db_server, [process_form/2]).
:- use_module(library('actmods/filebased_locate')).

main(_) :-
    get_form_input(Input),
    process_form(Input,Output),
    output_html(Output).
```

Phone DB Using Active Modules: Adding Phones

See `http://www.clip.dia.fi.upm.es/demo/pillow/phone_db_client2.cgi`

- Server active module (client is as before):

```
:- module(_,[process_form/2],[pillow]).
:- use_module(library(file_utils),[file_to_string/2]).
:- use_module(library(dynamic)).

process_form(Input,[cgi_reply|HTML_terms]) :-
    (  get_form_value(Input,input_name,IName),  \+ form_empty_value(IName)
    -> get_form_value(Input,phone_name,PName),  assert(phone(IName,PName)),
       Response = [ 'Added ', b(IName), ' / ',  b(PName), -- ]
    ;  get_form_value(Input,person_name,Name),  response(Name,Response) ),
    file_to_string('html_template2.html', Contents),
    html_template(Contents, HTML_terms, [response = Response]).

response(Name, []) :- form_empty_value(Name), !.
response(Name, ['Phone number for ',b(Name),' is ',Info, --]) :- phone(Name,Info), !.
response(Name, ['No phone number available for ',b(Name), '.', --]).

:- dynamic phone/2.
phone('CLIP',   '336-7448'). phone('Paco',   '554-5225'). phone('Daniel', '460-0569').
```

Phone DB Using Active Modules: Adding Phones w/Persistence

See `http://www.clip.dia.fi.upm.es/demo/pillow/phone_db_client2pers.cgi`

- Server active module (client is as before):

```
:- module(_,[process_form/2],[pillow,persdb]).
:- use_module(library(file_utils),[file_to_string/2]).
:- use_module(library(dynamic)).

process_form(Input,[cgi_reply|HTML_terms]) :-
    (  get_form_value(Input,input_name,IName), \+ form_empty_value(IName)
    -> get_form_value(Input,phone_name,PName),
        passertz_fact(phone(IName,PName)),
       Response = [ 'Added ', b(IName), ' / ', b(PName), hr$[] ]
    ;  get_form_value(Input,person_name,Name), response(Name,Response) ),
    file_to_string('html_template2.html', Contents),
    html_template(Contents, HTML_terms, [response = Response]).

response(Name, []) :- form_empty_value(Name), !.
response(Name, ['Phone number for ',b(Name),' is ',Info, --]) :- phone(Name,Info), !.
response(Name, ['No phone number available for ',b(Name), '.', --]).
```

Phone DB Using Active Modules: Adding Phones w/Pers (Cont.)

```
:- initialization(init_persdb).

:- multifile persistent_dir/2.
:- data persistent_dir/2.
persistent_dir(db,'./').

:- persistent(phone/2,db).
%% Database
phone('CLIP',   '336-7448').
phone('Paco',   '554-5225').
phone('Daniel', '460-0569').
```

Achieving Client-side ("Java-like") Functionality

- Automatic code downloading (client side processing):
  - Can be easily done for a particular browser (e.g., as a netscape "plug-in", or using Mosaic's API, as "LogicWeb").
  - Can actually be done independently of the browser! (see later)
- Supporting complex user interfaces (beyond forms):
  - Can be done e.g. using available tcl/tk "plug-ins".
  - Alternative: generate Java code from Prolog / use Java's graphical library.
- Also, use a Prolog to Java compiler: Bart Demoen's, Minerva, etc. (execution speed?).

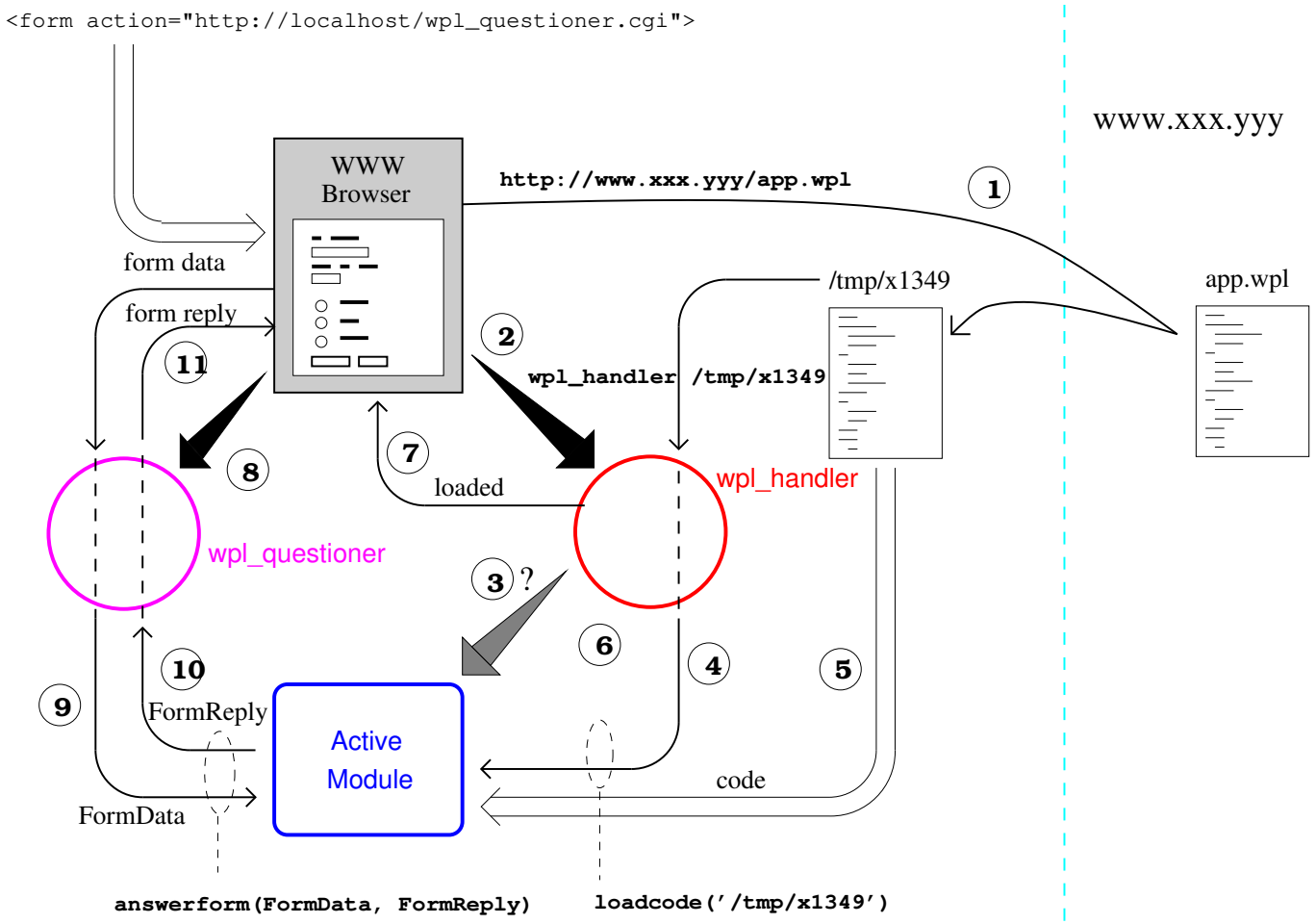
Automatic Code Downloading for Local Execution

- Using only the facilities presented, automatic LP/CLP code downloading for local execution is possible, using generic browsers.
- By simply clicking on a WWW pointer, and transparently for the user, remote code is automatically downloaded and locally queried via forms.
- Prerequisites:
  - The HTTP server on the server machine is configured to give a `mime.type` of `application/x-prolog` to the files with WWW-downloadable LP code.
  - The browser is configured to start helper `wpl_handler` when receiving data of type `application/x-prolog` (this application starts the LP engine as an active module).
  - There is a local cgi-bin executable `wpl_questioner.cgi` (which uses that active module).

Automatic Code Downloading Procedure

1. A click in a link of the query form starts the downloading of the code (alternatively it can also be done on page load using `multipart/mixed` mime type).
2. Browser starts a `wpl_handler` as document has type `application/x-prolog`.
3. The `wpl_handler` process starts a Prolog engine (configured as an active module) if necessary.
4. The handler asks the active module to read the code (trough a `loadcode(`$File$`)` call).
5. The active module reads the code and compiles it.
6. `wpl_handler` waits for active module to complete compilation, writes “done” to browser.
7. The browser receives the “done” message.
8. Pressing “submit” button in form now: browser starts a `wpl_questioner` as form handler.
9. The `wpl_questioner` process translates form data to a dictionary $FormData$, passing it to the active module through a call `answerform(`$FormData$`,`$FormReply$`)`.
10. Active module processes request, returns in $FormReply$ a WWW page (html term) containing answer.
11. The `wpl_questioner` process translates $FormReply$ to raw HTML and gives it back to the browser, dying afterwards. Subsequent queries proceed at 8.

Automatic Code Downloading Procedure – Figure



Higher-Level Models

Several models can be defined which provide a higher level of abstraction (e.g., higher level than PiLLoW):

- LogicWeb (Loke and Davison):
  - ◇ HTML pages can include Prolog code.
  - ◇ Any WWW page is seen by the Prolog code as a module. Module contains the page Prolog code plus some relations related to the HTML code.
  - ◇ Powerful module management.
  - ◇ Interesting applications shown.
- ALP ProWeb: provides persistence, also has templates, ...
- Other higher-level interfaces:
  - ◇ Generation of interfaces from database schemata [A.Porto]/RadioWeb,
  - ◇ WebDB: full database system with WWW interface (written in PiLLoW)

Some Other Work on LP/CLP + WWW

- PiLLoW includes previous work in `html.pl`, F. Bueno's WWW Chat version, and L. Naish's NU-Prolog forms.
- Also, K. Bowen's port of `html.pl` to ALS Prolog, which provides group processing of forms and an alternative to our use of active modules.
- Szeredi's multiple request handling through or-parallelism.
- ECLiPSe HTTP support library (by replacing HTTP server).
- Many LP/CLP Internet applications shown in recent workshops (many using PiLLoW)

Some Conclusions / Other Issues

- LP/CLP concepts/technology well suited for Internet applications – and exciting progress:
  - ◇ Many applications already developed ($WebChat$, Rent Advisor and others in the JICSLP'96 Workshop, many more now...).
  - ◇ Commercial systems are already providing interesting high-level functionalities.
- The $PiLLoW$ library has been designed to provide basic help in these tasks. It is available from: `http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`
- Many pointers can be found in the "CLIP/Compulog-Net LP/CLP and the WWW" Pages:

  `http://www.clip.dia.fi.upm.es/lpnet/index.html`
- Underlying support for **concurrency** and **distribution** (e.g., &-Prolog/Ciao, BinProlog/μ^2-Prolog, ...) has many advantages: e.g., overlapping requests

Some Conclusions / Other Issues (Contd.)

- Other interesting Internet/Distributed programming issues not covered:
  - VRML interfaces (e.g., ProVrml).
  - Blackboards and shared-variable based communication.
  - Agent programming.