

# CS-Prolog II

Version 2.3

## Graphical User Interface

ML Consulting and Computing Ltd.

Budapest, Hungary

May 1998



## Contents

1. Introduction .....	2
2. Programming environment .....	3
2.1 Projects .....	3
2.2 Generating code.....	4
2.3 Executing programs.....	4
2.4 Options .....	4
2.5 Starting the GUI environment .....	5
3. Multi-window trace .....	6
4. Web browser as CS-Prolog GUI component .....	7
4.1 Organization.....	7
4.2 Writing form handlers with the PiLLoW library .....	8
4.3 Generating HTML document from Prolog terms.....	9
4.3.1 General structures .....	9
4.3.2 Specific structures.....	11
4.4 Using PiLLoW in CS-Prolog II .....	13
4.5 Using the DCG converter .....	14

## 1. Introduction

CS-Prolog II is a console application that lacks any graphical user interface features. The CS-Prolog compiler, linker and runtime system get their arguments from the command line; all input/output is performed through files, including the standard input and standard output.

There are two important areas where this simple approach can be and must be improved. First is the programming environment; the second is a window-based graphical input/output for CS-Prolog user programs.

The most significant part of a programming environment is the debugging facility. In CS-Prolog several processes may run in parallel. Their trace output appears intermixed on the standard output making it difficult to sort out the information. So the main goal for the CS-Prolog programming environment is to implement a multi-window trace tool, where different processes send their debugging information to different windows. In the first version a simple project management is implemented, too, helping the user in working with programs consisting of several modules. The standard **make** utility is used with automatically generated makefiles. The programming environment is implemented in C++ and uses the Motif library.

The CS-Prolog input/output system is synchronous stream input/output, the predicates block the execution of the calling process. In a real-time application dealing with asynchronously incoming events this is not acceptable. Therefore we decided to separate the visual component implementing the graphical user interface and the CS-Prolog program itself. The GUI component interacts with the CS-Prolog applications using network communication.

We adopted the idea of the PiLLoW Library worked out by Daniel Cabeza, Manuel Hermenegildo and Sacha Varma at Computer Science Department of Technical University of Madrid (UPM), Spain. PiLLoW provides facilities for generating HTML structured documents, producing HTML forms, and writing form handlers, thus making possible to use a web browser for GUI purposes. One drawback of this approach is that a web server is needed on the network.

The work on the extensions described in this supplement had been partially funded by the EU in the framework of the INCO-Copernicus project *Expernet: A Distributed Expert System for the Management of a National Network*, No 960114.

## 2. Programming environment

The main window of the programming environment consists of a menu, a text box where the output information is displayed and, at the bottom of the window, the module list box where the module names of the current project are enumerated.

The main menu of CS-Prolog programming environment consists of the following menu elements:

- File
- Project
- Execute
- Make
- Options
- About

The **File** submenu contains a single operation: **Exit**, to quit the environment.

The **Project** submenu contains the operations for handling CS-Prolog projects: creation of a new project, opening of an existing one, deleting a project, adding and removing source modules to and from the project, and specifying local options.

The **Execute** submenu operations serve for executing and tracing CS-Prolog programs.

The **Make** submenu contains operations for compiling and linking the source modules, removing the generated files.

With **Options** submenu elements the user can specify parameter settings for the current project or globally for any new project.

### 2.1 Projects

The basic object handled in the CS-Prolog II programming environment is the **project**. Projects consist of CS-Prolog modules that form the CS-Prolog program and they accommodate some other information as: compiler options, runtime command line options, header file dependencies, break points, etc.

Projects are created with the **New** menu operation; modules have to be added with **Add module** menu element. The file names of a project's modules are enumerated in a list box at the bottom part of the main window. If in this list box a module is selected then this module can be deleted with the **Remove module** menu item. Local compilation options can be assigned to a module with **Local options ...** menu item.

The environment finds out the dependency structure of a project, i.e., which *include* files a specific module depends on. This dependency information is utilized when creating the *makefile* that generates the program (see next section). Suppression of inclusion of an *include* file into the makefile can be done with **Include files ...** menu item.

A project can be saved explicitly with **Close** operation and it is automatically saved if a new project is created or another project is opened with the **Open** operation.

## 2.2 Generating code

The environment creates a makefile based on the project information. The standard Unix **make** utility can read this makefile and generate the compiled **mdf** and linked **pdf** files.

From the programming environment the code generation can be initiated with the **Build** element of the **Make** submenu. Only those files will be regenerated that are out of date (as it is usual with the **make** utility). The **Rebuild All** menu item will regenerate all files, even those that are up to date. The **Compile** and **Link** menu items are used for the compilation of a module and linking the program, respectively.

The **Clean** menu element has a submenu itself; the user can choose between cleaning all generated files, or cleaning only the intermediate files, preserving the executable CS-Prolog program.

The makefile can be used outside the CS-Prolog environment as well. Its name is

```
<project_name>.mak
```

The targets for the **make** utility using this makefile can be: **build**, **rebuild**, **clean**, **realclean** and **exec**. These targets correspond to the environment operations explained above, (the **exec** target will execute the program).

## 2.3 Executing programs

The **Execute** main menu element consists of three operations: **Run** (execute without debugging), **Trace** (execute with debugging) and **Dialog...** (execute via a dialog box). In the latter operation the user can specify command line options for the program to be executed or traced.

If a program is always run with the same command line arguments then these arguments can be hardwired setting an execute option (see next section). In this case the program can be executed or traced without having to type in explicitly the arguments.

The executed program inherits standard input and standard output from the terminal where the environment was called from. So if the program writes out to or reads in from the standard stream, it appears on the **stdout** or must be typed in the **stdin** of the original terminal.

The trace system of the programming environment is discussed in chapter 3.

## 2.4 Options

The **Option** main menu element has two submenus: **Project options** and **Workspace options**. Both offer three further operations: **Compile options...**, **Build options...** and **Execute options...** Workspace options are the settings that will be default for any newly created project. These default options can be changed later with **Project options** operations.

In the compiler options dialog box the user can set the option values for the CS-Prolog compiler. In build options dialog the names of generated files can be changed, and traced code generation can be switched off.

In execute options dialog box the user can set the name for the CS-Prolog runtime system (if it is not the default one) and specify command line argument for the execution.

The workspace option submenu contains one more item: the **Customize...** action. In the customization dialog box the user can set properties of the programming environment itself. Most of these properties are represented by toggle buttons; the following features can be switched on or off:

- automatically build the project if it is out of date and it is executed;
- automatically rebuild the project if it was built without trace and now it is executed with trace;
- save debug breakpoints when the project is closed;
- display the module list box at the bottom of the main window;
- write verbose messages to the information text box;
- ask for confirmation if a project is deleted;
- ask for confirmation if a Quit trace action is initiated;
- ask for confirmation if a trace window is closed;
- automatically load the last project on startup or automatically load a project file if it is unique in the current working directory.

The last property is the *menu bar dimming*. By default the environment disables its main menu when an external action is performed: program building or program execution. The user can set the time delay of the dimming, or can switch off this feature.

## 2.5 Starting the GUI environment

The executable normally resides in the home directory of CSP-II. It can be started from **xterm** using for example the following console command:

...

All the usual xterm environment arguments can be used in the command. The default background color is blue. If more than one application is being debugged then it is a good idea to use different background colors for each (trace windows inherit the background from the main window).

### 3. Multi-window trace

The most significant subsystem of the CS-Prolog programming environment is the multi-window trace. It offers a menu-driven tracing window where the trace information is displayed. The different CS-Prolog processes have independent trace windows, so their debugging information is separated from each other (see also the chapter on debugging in the User's Manual).

A trace window appears on the screen whenever a process begins its execution. Initially the trace window of the main process is displayed. Trace windows have a main menu with the following items. (Each menu item represents an action that is available in normal text based debugging described in CS-Prolog User's Manual.)

Trace

Single step execution. Trace will stop at the entry port of the predicate called next.

Skip

Skip the trace of the current call. Trace will stop at the entry port of the predicate called next, after the termination (successful or failed) of the current call.

Quit

Abort the debugging and the execution.

Go

Go without trace. Trace will stop at the entry port of the next break point.

Prolog

Enter a Prolog session. CS-Prolog goals can be executed. This session however takes place outside the environment, on the terminal where the environment was started. (The session can be finished by entering the **halt .** command.)

Breakpoint

Set or clear break points in a dialog box. The breakpoints can be set in two modes: selecting the predicate in a list box, or writing explicitly the functor. In the first case however only the statically compiled predicates are enumerated in the list box.

Close

Close the trace window. The execution will continue, but the process this trace window was associated with will not be debugged from now on.

## 4. Web browser as CS-Prolog GUI component

Real-time programs that react to asynchronous events should not be allowed to perform input/output operations that can block the application for a long time. The basic input/output predicates of CS-Prolog, however, are blocking the caller application on mono-processor hosts, e.g., when manual input is awaited from an operator. The solution for this problem can be the separation of the application component which interacts with the user from the real-time components, and organizing communication between them using network connection.

The idea of using a web browser as a graphical user interface component for CS-Prolog was suggested by Manuel Hermenegildo. He and his colleagues at the Technical University of Madrid (UPM) implemented a Prolog library named PiLLoW that provides facilities for generating HTML documents (pages) and handling HTML forms. The PiLLoW system is described in details at the following URL:

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

The web browser can display HTML documents containing HTML forms where the user specifies his/her data to be sent to the CS-Prolog application. When the form is *submitted* the browser executes the associated *form handler* — a shell script, which can read the HTML form definition from the standard input and is expected to produce a reply HTML document on standard output displayed by the browser as a new document. The script in its turn may launch a (prolog) program capable of communicating with the distributed CS-Prolog application for composing the reply. The launched program is called the *gateway* program for short. This scenario assumes a *form-capable* browser, i.e., one that can handle input from HTML forms.

Another scenario of interactive work with web browser is to update the content of some HTML page or pages directly from the application and have the browser refresh the display at an appropriate rate when the page is shown (if the browser supports refreshing). This method is suited for displaying alerts or monitoring system status.

The two approaches can be combined, and, of course, simple applications can use the PiLLoW library directly also in the request-reply method, i.e., the gateway program can be the application itself.

This specification cannot provide a comprehensive introduction to HTML, so a familiarity with the topic is supposed on the part of the reader. The following sections are based on the original PiLLoW documentation quoted above.

### 4.1 Organization

The general request-reply scenario requires several preparatory steps be performed by the application developer. First the initial HTML page has to be written which is the starting point for the user interaction (or a script dynamically composing the page). This initial page should include one or more forms for user input. Each form can contain text fields, check boxes, radio buttons and menus. The operator will formulate a question by setting these controls to appropriate values. Every form can contain a **Submit** push-button. When the user clicks this button (or hits the **Enter** key in some cases) the content of the form is sent to a so called CGI executable associated with the button (CGI stands for Common Gateway Interface). This script can start other programs – in our

case a CS-Prolog program, which will receive the content of the HTML form; let's call this application the *gateway* program. The gateway program is probably specific to the application (there can be several of them); it is the agent which understands both the HTML particulars and the notions the application reveals to the operator.

The input control settings are analyzed by the gateway program, which connects through the network to the target real-time CS-Prolog application. When the answer for the question formulated by the user in the HTML form is negotiated, a new HTML document constituting the answer is generated by the gateway program. Of course a new HTML form can be included into this document too, where the user can continue the communication with the application. It is also a good idea to place a link back to the initial document on the answer page. When the formation of the answer HTML document is completed the gateway program should terminate.

## 4.2 Writing form handlers with the PiLLoW library

The PiLLoW library provides the following predicates to simplify the task of getting the input from a form:

```
get_form_input(Dic)
```

```
get_form_input(S_or_a, Dic)
```

Translate input from the form to a dictionary **Dic** (as a list of **attribute=value** pairs). It translates empty values (which indicate only the presence of an attribute) to '**\$empty**', values with more than one line (from text areas or files) to a list of lines as strings, the rest to atoms or numbers. Input is read from the stream specified by the **S\_or\_a** argument.

```
get_form_input(Dic)
```

is equivalent with

```
current_input(Stream), get_form_input(Stream, Dic)
```

```
get_form_value(Dic, Var, Val)
```

Gets value **Val** for attribute **Var** in dictionary **Dic**. Does not fail: value is '' (the empty atom) if **Var** is not found.

```
text_lines(Val, Lines)
```

Transforms a value given by a dictionary to a list of lines, for data coming from a text area.

```
form_empty_value(V)
```

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

```
form_default(Val, Default, NewVal)
```

Useful when a form is only partially filled. If the value of **Val** is empty then **NewVal=Default**, else **NewVal=Val**.

## 4.3 Generating HTML document from Prolog terms

HTML documents can be easily composed in Prolog by directly writing the text and the necessary markup properly arranged to any output stream. This method is, however, somewhat inelegant and the HTML-specific marks render the source code hard to read. PiLLoW offers the possibility to have an encoding of HTML code as Prolog terms, which could be manipulated easily, and a predicate to translate such terms to HTML for output.

The basic predicate in PiLLoW to provide the functionality of translating Prolog terms to HTML code is:

```
output_html(S_or_a, F)
```

It accepts in **F** a Prolog HTML term or a list of HTML terms and sends to the stream specified by the **s\_or\_a** argument the text which is the rendering of the term(s) in HTML format. The one-argument variant

```
output_html(F)
```

Is equivalent with

```
current_output(Stream), output_html(Stream, F)
```

**output\_html** will occasionally fail if it encounters a structure which is not a legal HTML term.

HTML is a quite powerful language. For using its full capabilities PiLLoW provides general recursive structure manipulations. For simple cases, however, more intuitive special structures are also provided.

In a HTML term certain structures and atoms represent special functionality (HTML marking). An HTML term can be recursively a list of HTML terms. The following are legal HTML terms:

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

When converting HTML terms to character (HTML document) format, PiLLoW translates these structures recursively. Strings are always left unchanged. HTML terms must be fully instantiated (ground) at the moment when the term is going to be translated. This allows creating documents piecemeal, backpatching of references in documents, etc.

In the following sections we list the meaning of the principal Prolog structures that represent special HTML functionality. Only special atoms are translated; the rest are assumed to be normal text and will be passed into the HTML document unchanged.

### 4.3.1 General structures

Basically, HTML has two kinds of components: HTML *elements* and HTML *environments*.

An HTML element has the form

```
<NAME Attributes>
```

where **NAME** is the name of the element and **Attributes** is a (possible empty) sequence of attributes, each of them being either an attribute name or attribute assignment of the form

```
attrname="Value".
```

An HTML environment has the form

```
<NAME Attributes> Text </NAME>
```

where **NAME** and **Attributes** are the same as above, **Text** represents data specific to the particular marking (specified by **NAME**).

The general Prolog structures are used for representing these two HTML constructs are:

```
Name $ Atts
```

Represents an HTML element of name **Name** and its attributes **Atts** (**\$/2** is defined as an infix binary operator). For example, the term

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

is translated into the HTML construct

```

```

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

```
name(Text)
```

(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 construct

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

```
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')
```

is translated into the HTML construct

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

```
env(Name, Atts, Text)
```

Equivalent with `Name(Atts, Text)`.

```
begin(Tag)
```

```
begin(Tag, Atts)
```

These translate to the start of an HTML environment of name **Tag** and attributes **Atts** (second form). Useful in conjunction with the next structure, when a piece of separately prepared code is to be inserted into the document (for example, **Tag** is **pre**). In other circumstances their usage is discouraged.

```
end(Tag)
```

Translates to the end of an HTML environment of name **Tag**.

Any HTML constructs can be represented with these structures (except comments and declarations, which can be inserted as atoms or strings), but the PiLLOW library provides additional, specific constructs to simplify HTML creation.

### 4.3.2 Specific structures

We list below the most important special structures for HTML that PiLLOW understands:

`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>`).

`title(Title)`

Used for defining the document title which is usually shown in the caption by the browsers (translates to `<title>"Title"</title>`).

`title(Title, Refresh_rate)`

Used for defining the document title as in the case above, and also to specify a time interval in seconds for the browser as the period for refreshing the displayed page from its source document. Refreshing is useful when the underlying document is dynamically changed by a running application for alerts or monitoring (translates to `<title>"Title" <meta http-equiv="refresh" content="Refresh_rate"></title>`). The use of this form is not advisable for 'reply' type documents.

`comment(Comment)`

Inserts an HTML comment (translates to `<!-- Comment -->`).

`declare(Decl)`

Inserts an HTML declaration — seldom used (translates to `<!Decl>`).

`image(Addr)`

Used for inserting an image at address (URL) **Addr** (translates to an `<img>` element).

`image(Addr, Atts)`

As above, 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 **<hN>** environment).

itemize(*Items*)

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

enumerate(*Items*)

Produces a list of numbered items, **Items** is a list of corresponding HTML terms (translates to an **<ol>** environment).

description(*Defs*)

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

nice\_itemize(*Img*, *Items*)

Produces a list of bulleted 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 others (if any) are defined terms (translates to a **<dl>** environment).

preformatted(*Text*)

Used to include preformatted text. **Text** is a list of HTML terms, each element of the list being a line of the resulting documents with layout characters preserved (translates to a **<pre>** environment).

entity(*Name*)

Includes the entity of name **Name** (ISO-8859-1 special character).

verbatim(*Text*)

Used to insert text verbatim. Special HTML characters (**<**, **>**, **&**, **"**) are translated to their quoted equivalent.

nl

Used to insert a newline into the HTML document (just to improve human readability).

cgi\_reply

This is not HTML; the CGI protocol requires this content descriptor to be used by CGI executables (including form handlers) when replying (translates to **<content-type: text/html>**).

pr

Inserts into the page a graphical logo with the message

**Developed using the PiLLOW Web programming library**

which also points to the manual and library source.

The HTML terms used for creating input forms inside a document, are the following:

start\_form(*Addr*, *Atts*)

Specifies the beginning of a form. **Addr** is the address (URL) of the program that will handle the form, and **Atts** contains other attributes of the form (translates to **<form action="Addr" Atts>**).

`start_form(Addr)`

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

`start_form`

Specifies the beginning of a form without assigning address to the handler, so that the form handler will be the cgi-bin executable producing the form (translates to `<form>`).

`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, otherwise **State=off** (translates to an `<input>` element).

`radio(Name, Value, Selected)`

Specifies an input of type radio with name **Name** (the radio buttons that are interlocked must share their name), **Value** is 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**, **hidden**, **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).

There is also a special predicate `html_expansion/2` predicate provided for defining new structures. It is a dynamic partition of the `pillow` module (changed so for CSP-II), clauses for translating additional user-defined specific structures can be appended to it using the `assertz` built-in predicate. The head of `html_expansion/2` is the following:

```
html_expansion(UserTerm, CanonicTerm)
```

where `UserTerm` is the additional term being defined and `CanonicTerm` is the definition using already known HTML terms. Care must be taken in order not to create infinite recursion (for example, by defining a new term with itself).

## 4.4 Using PiLLOW in CS-Prolog II

CS-Prolog II uses a slightly modified version of the PiLLOW library. At the interface level the differences are in the addition of `get_form_input/2` and `output_html/2` predicates that allow the user to specify any source and sink, respectively, in place of the standard streams, and the extended variant of the `title` HTML term for specifying a refresh rate. Another difference is the omission of the `fetch_url/3` predicate. The last difference is that `html_expansion/2` is made dynamic, additional clauses can be appended to it using `assertz`.

For the description of other PiLLOW predicates not covered here see the original PiLLOW manual at the place indicated earlier.

The compiled library is placed in the home directory as `pillow.mdf` along with `standard.mdf` and `csprolog.mdf`, but, unlike the latter ones, it has to be linked explicitly with the other compiled modules (complete path specification is necessary) of the application when the `.pdf` file is created. The name of the module is `pillow`.

There is also a header file named `pillow.inc` in the home directory that can be included into the source module for specifying the imported PiLLOW predicates, or used as an example for the necessary **import** directives. The following example shows how the full set of exported predicates can be imported using the header file:

```
import(  
  #include <pillow.inc>  
).
```

Besides, if the general structures are used, then '\$' has to be declared as an infix operator, both compile time and runtime, as for example:

```
:-op(150, xfy, '$').
```

among the directives, and

```
op(150, xfy, '$').
```

somewhere in the initial phase.

## 4.5 Using the DCG converter

The original PiLLOW library source uses DCG syntax. CSP-II lacks this feature, so a separate DCG converter program had been developed for adapting the library. The converter is based on a DCG compiler written in Edinburgh by Fernando Pereira, EDCAAD, in 1984.

The converter can be used for converting any other CSP-II source containing DCG clauses. In order to do this, first the source code has to be preprocessed (unless it is pure wrt. to the preprocessor). The converter itself acts as a filter, reading input from the standard input and writing the result to the standard output. Here is an example showing how the PiLLOW library had been converted (after making other syntactical adjustments manually):

```
cspcomp -P pillow  
csprolog -cspprog convdgc <pillow.i >dpillow.pro  
cspcomp -fo pillow.mdf dpillow
```

The converter itself is made of two modules: **dcg.pro** and **convdgc.pro**; these sources can be found in the **examples** subdirectory of the distribution. The runnable **convdgc.pdf** is placed into the **utilities** subdirectory.