SWI-Prolog Source Documentation
Version 2

Jan Wielemaker
VU University, Amsterdam
The Netherlands
E-mail: J.Wielemaker@vu.nl

January 12, 2018

Abstract

This document presents PlDoc, the SWI-Prolog source-code documentation infrastructure. PlDoc is loosely based on JavaDoc, using structured comments to mix documentation with source-code. SWI-Prolog's PlDoc is entirely written in Prolog and well integrated into the environment. It can create HTML+CSS and LaTeX documentation files as well as act as a web-server for the loaded project during program development.

The SWI-Prolog website (http:www.swi-prolog.org) is written in Prolog and integrates PlDoc to provide a comprehensive searchable online manual.

Version 2 of PlDoc extends the syntax with Markdown markup as specified by Docygen. Based on experience with version 1, PlDoc 2 both tightens some rules to avoid misinterpretations and relaxes others that were considered too conservative.

Contents

1 Introduction

When developing Prolog source that has to be maintained for a longer period or is developed by a —possibly distributed— team some basic quality mechanisms need to be adopted. A shared and well designed codingstyle [Covington *et al.*, 2012] is one of them. In addition, documentation of source-files and their primary interfaces as well as a testing framework must be established.

In our view, hitherto existing documentation and testing frameworks fell short realising the basic needs in a lightweight and easy to adopt system. To encourage consistent style, well commented code and test-assisted development, we make sure that

- The documentation and testing framework requires a minimum of work and learning.
- The framework is immediately rewarding to the individual programmer as well as the team,

First, we describe the documentation system we developed for SWI-Prolog. In section 11 we motivate our main choices.

2 Overview

Like JavaDoc, the PlDoc infrastructure is based on *structured comments*. Using comments, no changes have to be made to Prolog to load the documented source. If the `pldoc` library is loaded, Prolog will not only load the source, but also parse all structured comments. It processes the mode-declarations inside the comments and stores these as annotations in the Prolog database to support the test framework and other runtime and compiletime analysis tools that may be developed in the future.

Documentation for all or some of the loaded files can be written to file in either HTML+CSS or LaTeX (see section 10.5) format. Each source file is documented in a single file. In addition, the documentation generator will generate an index file that can be used as an index for a browser or input file for LaTeX for producing nicely typeset document.

To support the developer, the documentation system can be asked to start a web-server that can be used to browse the documentation.

3 Structured comments

Structured comments come in two flavours, the line-comment (%) based one, seen mostly in the Prolog community and the block-comment (`/*...*/`) based one, commonly seen in the Java and C domains. As we cannot determine the argument names, type and modes from following (predicate) source itself, we must supply this in the comment.[1] The overall structure of the comment therefore is:

- Semi-formal type- and mode-description, see section 5
- Wiki-style documentation body, see section 7
- JavaDoc style tags (`@`keyword value, see section 6)

The `/*...*/` style comment starts with `/**`¡white¿. The type and mode declarations start at the first non-blank line and are ended by a blank line.

[1] See section 11.

The %-style line comments start with %!¡white¿ or, for compatibility reasons, with %%¡white¿.[2] The type and mode declaration is ended by the first line that starts with a single %. E.g., the following two fragments are identical wrt. PlDoc. Skipping blank-lines in /** comments allows to start the comment on the second line.

```
%!      predicate(-Arg:type) is nondet
%       Predicate ...
```

```
/**
 * predicate(-Arg:type) is nondet
 *
 * Predicate ...
 */
```

The JavaDoc style keyword list starts at the first line starting with @⟨*word*⟩.

4 File (module) comments

An important aspect is documentation of the file or module as a whole, explaining its design, purpose and relation to other modules. In JavaDoc this is the comment that preceeds the class definition. The Prolog equivalent would be to put the module comment in front of the module declaration. The module declaration itself however is an important index to the content of the file and is therefore best kept first.

The general comment-structure for module comments is to use a type identifier between angled brackets, followed by the title of the section. Currently the only type provided is `module`. Other types may be added later.

Example

```
/** <module> Prolog documentation processor

This module processes structured comments and generates both formal
mode declarations from them as well as documentation in the form of
HTML or LaTeX.

@author Jan Wielemaker
@license GPL
*/
```

[2]The %% leader was considered to give too many false positives on arbitrary source code. It is still accepted, but invalid comments are silently ignored, while invalid comments that start with %! result in a warning.

5 Type, mode and determinism declaration headers

Many predicates can sensibly be called in different ways, e.g. with a specific argument as input or as output. The header of the documentation of a predicate consists of one or more *templates*, each representing a specific way of calling the predicate.

A template can contain information about types, argument instantiation patterns, determinism and more. The syntax is informally described below:

⟨*template*⟩	::=	⟨*head*⟩['//'] 'is' ⟨*determinism*⟩
	|	⟨*head*⟩['//']
⟨*determinism*⟩	::=	'det'
	|	'semidet'
	|	'failure'
	|	'nondet'
	|	'multi'
⟨*head*⟩	::=	⟨*functor*⟩'('¡argspec¿ ',' ⟨*argspec*⟩')'
	|	⟨*functor*⟩
⟨*argspec*⟩	::=	[⟨*instantiation*⟩]⟨*argname*⟩[':'¡type¿]
⟨*instantiation*⟩	::=	'++' | '+' | '-' | '–' | '?' | ':' | '@' | '!'
⟨*type*⟩	::=	⟨*term*⟩

The *determinism* values originate from Mercury. Their meaning is explained in the table below. Informally, `det` is used for deterministic transformations (e.g. arithmetic), `semidet` for tests, `nondet` and `multi` for *generators*. The `failure` indicator is rarely used. It mostly appears in hooks or the recovery goal of `catch/3`.

Determinism	**Predicate behaviour**
`det`	Succeeds exactly once without a choice point
`semidet`	Fails or Succeeds exactly once without a choice-point
`failure`	Always fails
`nondet`	No constraints on the number of times the predicate succeeds and whether or not it leaves choice-points on the last success.
`multi`	As `nondet`, but succeeds at least one time.

The meanings of the *instantiation patterns* for individual arguments are:

++	Argument is ground at call-time, i.e., the argument does not contain a variable anywhere.
+	Argument is fully instantiated at call-time, to a term that satisfies the type. This is not necessarily *ground*, e.g., the term `[_]` is a *list*, although its only member is unbound.
-	Argument is an *output* argument. It may be unbound at call-time, or it may be bound to a term. In the latter case, the predicate behaves as if the argument was unbound, and then unified with that term after the goal succeeds. For example, the goal `findall(X, Goal, [T])` is good style and equivalent to `findall(X, Goal, Xs), Xs = [T]`[3] Determinism declarations assume that the argument is a free variable at call-time. For the case where the argument is bound or involved in constraints, `det` effectively becomes `semidet`, and `multi` effectively becomes `nondet`.
–	Argument is unbound at call-time. Typically used by predicates that create 'something' and return a handle to the created object, such as `open/3` which creates a *stream*.
?	Argument is bound to a *partial term* of the indicated type at call-time. Note that a variable is a partial term for any type.
:	Argument is a meta-argument. Implies +.
@	Argument will not be further instantiated than it is at call-time. Typically used for type tests.
!	Argument contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.

Users should be aware that calling a predicate with arguments instantiated in a way other than specified by one of the templates may result in errors or unexpected behavior.

Developers should ensure that predicates are *steadfast* with respect to output arguments (marked - in the template). This means that instantiation of output arguments at call-time does not change the semantics of the goal (it may be used for optimization, though). If this steadfast behavior cannot be guaranteed, – should be used instead.

In the current version, argument *types* are represented by an arbitrary term without formal semantics. In future versions we may adopt a formal type system that allows for runtime verification and static type analysis [Hermenegildo, 2000, Mycroft & O'Keefe, 1984, Jeffery *et al.*, 2000]

Examples

```
%!      length(+List:list, -Length:int) is det.
%!      length(?List:list, -Length:int) is nondet.
%!      length(?List:list, +Length:int) is det.
%
%       True if List is a list of length Length.
%
%       @compat iso
```

6 Tags: @see, etc.

Optionally, the description may be followed by one or more *tags*. Our tag convention is strongly based on the conventions used by javaDoc. It is adviced to place tags in the order they are described below.

@**arg** *Name Description*
: Defines the predicate arguments. Each argument has its own `@arg` tag. The first word is the name of the argument. The remainder of the tag is the description. Arguments declarations normally appear in order used by the predicate.

@**param** *Name Description*
: This is a synonym for `@arg`, using the JavaDoc tag name.

@**throws** *Term Description*
: Error condition. First Prolog term is the error term. Remainder is the description.

@**error** *Error Description*
: As @throws, but the exception is embedded in `error`(*Error, Context*).

@**author** *Name*
: Author of the module or predicate. Multiple entries are used if there are multiple authors.

@**version** *Version*
: Version of the module. There is no formal versioning system.

@**see** *Text*
: Point to related material. Often contains links to predicates or files.

@**deprecated** *Alternative*
: The predicate or module is deprecated. The description specifies what to use in new code.

@**compat** *Standards and systems*
: When implementing libraries or externally defined interfaces this tag describes to which standard the interface is compatible.

@**copyright** *Copyright holder*
: Copyright notice.

@**license** *License conditions*
: License conditions that apply to the source.

@**bug** *Bug description*
: Known problems with the interface or implementation.

@**tbd** *Work to be done*
: Not yet realised behaviour that is enticipated in future versions.

7 Wiki notation

Structured comments that provide part of the documentation are written in Wiki notation, based on TWiki, with some Prolog specific additions.

7.1 Structuring conventions

Paragraphs Paragraphs are seperated by a blank line. Paragraphs that are indented in the source-code *after* normalising the left-margin are also indented in the output. Indentation is realised in the HTML backend using a `blockquote` element and in LaTeX using the `quote` environment. Finally, if the initial indentation is 16 or more, the paragraph is *centered*.

General lists The wiki knows three types of lists: *bullet lists* (HTML `ul`), *numbered lists* (HTML `ol`) and *description lists* (HTML `dl`). Each list environment is headed by an empty line and each list-item has a special symbol at the start, followed by a space. Each subsequent item must be indented at exactly the same column. Lists may be nested by starting a new list at a higher level of indentation. The list prefixes are:

`*`	Bulleted list item
`1.`	Numbered list item. Any number from 1..9 is allowed, which allows for proper numbering in the source. Actual numbers in the HTML or LaTeX however are re-generated, starting at 1.
`$ Title : Item`	Description list item.

Term lists Especially when describing option lists or different accepted types, it is common to describe the behaviour on different terms. Such lists must be written as below. ⟨*Term1*⟩, etc. must be valid Prolog terms and end in the newline. The Wiki adds `' . '` to the text and reads it using the operator definitions also used to read the mode terms. See section 5. Variable names encountered in the *Term* are used for indentifying variables in the following *Description*. At least one *Description* must be non-empty to avoid confusion with a simple item list.

```
* Term1
  Description
* Term2
  Description
```

Predicate description lists Especially for processing Wiki files, the Wiki notation allows for including the description of a predicate 'in-line', where the documentation is extracted from a loaded source file. For example:

```
The following predicates are considered Prolog's prime list processing
primitives:

    * [[member/2]]
    * [[append/3]]
```

Tables The Wiki provides only for limited support for tables. A table-row is started by a `|` sign and the cells are separated by the same character. The last cell must be ended with `|`. Multiple lines that parse into a table-row together form a table. Example:

```
        | Algorithm    | Time (sec) |
        | Depth first  | 1.0        |
        | Breath first | 0.7        |
        | A*           | 0.3        |
```

Section Headers Section headers are creates using one of the constructs below taken from TWiki. Section headers are normally not used in the source-code, but can be useful inside README and TODO files. See section 8.

```
---+ Section level 1
---++ Section level 2
---+++ Section level 3
---++++ Section level 4
```

In addition, PlDoc recognises the *markdown* syntax, including named sections as defined by `doxygen`. A section is named (labeled) using an optional sequence `{\#`*name*`}`. The three code sections below provide examples. Note that `#` section headers should be positioned at the left margin and the `#` must be followed by blank space. If the header is underlined, the underline is a line that only contains = or − characters. There must be a minimum of three[4] of such characters.

```
Section level 1
===============

Section level 2
---------------
```

```
# Section level 1
## Section level 2
### Section level 3
#### Section level 4
```

```
Section level 1         {#label}
===============

# Section level 1       {#label}
```

Code blocks There are two ways to write a code block. The first one is *fenced*. Here, the block is preceeded and followed by a fence line. The traditional PlDoc fence line is ==. Doxygen fence lines are also accepted. They contain at least three tilde (~) characters, where the opening fence line may be followed by a file extension between curly brackets. In all cases, the code is indented relative to the indentation of the fence line. Below are two examples, the first being the

[4] Markdown demands two, but this results in ambiguities with the == fence for code blocks.

traditional PlDoc style. The second is the Doxygen style, showing a code block that is indented (because it is a body fragment) and that is flagged as Prolog source. Note that the `{.pl}` is optional.

```
==
small(X) :-
   X < 2.
==
```

```
~~~{.pl}
  ...,
  format('Hello ~w~n', [World]),
  ...,
~~~
```

The second form of code blocks are *indented blocks*. Such a block must be indented between 4 and 8 characters, relative to the indentation of the last preceeding non-blank line. The block is opened with a blank line and closed by a blank line or a line that is indented less than the indentation of the initial line. It is allowed to have a single blank line in the middle of a code block, provided that the next line is again indented at least as much as the initial line. The initial line as well as a line that follows a blank line may not be a valid list opening line or a table row, i.e., it may not start with one of `*-` followed by a space or `|`.

Rulers PlDoc accepts both the original PlDoc and markdown conventions for rulers. A PlDoc ruler is a line with at least two dashes (-) that starts at the left-most column. A markdown ruler holds at least three ruler characters and any number of spaces. The ruler characters are the dash (-), underscore (_) or asterisk (*). Below are three examples, the last two of which are valid markdown.

```
--
***
                - - -
```

Line breaks A line break may be added by *ending* the physical line with the HTML linebreak, `<br>` or `<br/>`.[5]

7.2 Text markup: fonts and links

7.2.1 Emphasizing text

Text emphasis is a combination of old plaintext conventions in Usenet and E-mail and the doxygen version of markdown. Table 1 shows the font-changing constructions. The phrase *limited context* means that

[5]The markdown conventions are (original) two spaces at the of the physical line and (GitHub) a physical line break. Neither fit well with source code. Doxygen supports restricted HTML and allows for `<br>`.

`*bold*`	Typeset text in **bold** for limited content (see running text).
`*|bold|*`	Typeset text in **bold**. Content can be long.
`_emphasize_`	Typeset text as *emphasize* for limited content (see running text).
`_|emphasize|_`	Typeset text as *emphasize*. Content can be long.
`=code=`	Typeset text `fixed` font for identifiers (see running text).
`=|code|=`	Typeset text `fixed` font. Content can be long.
`Word`	Capitalised words that appear as argument-name are written in *Italic*

Table 1: Wiki constructs to change the font

- The opening `*` or `_` must be preceeded by white space or a character from the set `<{([,:;` and must be followed by an alphanumerical character.
- The closing `*` or `_` may not be followed by an alphanumerical character and may not be preceeded by white space or a character from the set `({[<=+-\@`.
- The scope of these operations is always limited to the identified structure (paragraph, list item, etc.)

Note that =¡identifier¿= is limited to a an *identifier*, such as a file name, XML name, etc. Identifiers must start and end with an alphanumerical character, while characters from the set `.-/:` may appear internally. Note that this set explicitly does not allow for white space in code spans delimited by a single =. This markup is specifically meant to deal with code that is either not Prolog code or invalid Prolog code. Valid Prolog code should use the backtick as described in section 7.2.2.

7.2.2 Inline code

Inline code can be realised using the = switch described in section 7.2.1 or the markdown backtick. In addition, it can use the mardown/Doxygen *backtick* (`) convention: a string that is delimited by backticks is considered code, provided:

- An internal double backtick is translated into a single backtick.
- Inline code is limited to the current structure (paragraph, table cell, list item, etc.
- The content of the code block is valid Prolog syntax. Note that in Doxygen, the syntax is not validated and a single quote cancels the recognition as code. The latter is a problematic in Prolog because single quotes are often required.

Currently, 'Var' is typeset as a variable (italics) and other terms are typeset using a fixed-width code font.

In addition, compound terms in canonical notation (i.e., *functor*`(`*,...args...*`)` that can be parsed are first verified as a file-specification for `absolute_file_name/3` and otherwise rendered as *code*.

7.2.3 Links

Table 2 shows the constructs for creating links.

`name/arity`	Create a link to a predicate
`name//arity`	Create a link to a DCG rule
`name.ext`	If ⟨*name*⟩.⟨*ext*⟩ is the name of an existing file and ⟨*ext*⟩ is one of `.pl`, `.txt`, `.md`, `.png`, `.gif`, `.jpeg`, `.jpg` or `.svg`, create a link to the file.
`prot://url`	If ⟨*prot*⟩ is one of `http`, `https` or `ftp`, create a link.
`<url>`	Create a hyperlink to URL. This construct supports the `expand_url_path/2` using the construct ⟨*alias*⟩:⟨*local*⟩. ⟨*local*⟩ can be empty.
`[[label][link]]`	Create a link using the given ⟨*label*⟩. Label can be text or a reference to an image file. Additional arguments can be supplied as `;`⟨*name*⟩`="`⟨*value*⟩`"`. More arguments are separated by commas. ⟨*link*⟩ must be a filename as above or a url.
`[label](link)`	The markdown version of the above.

Table 2: Wiki constructs that create links

7.3 Images

Images can be included in the documentation by referencing an image file using one of the extensions `.gif`, `.png`, `.jpeg`, `.jpg` or `.svg`.[6] By default this creates a link to the image file that must be visited to see the image. Inline images can be created by enclosing the filename in double square brackets. For example

```
The [[open.png]] icon is used open an existing file.
```

The markdown alternative for images is also supported, and looks as below. The current implementation only deals with image *files*, not external resources.

```
    ![Caption](File)
```

8 Directory indices

A directory index consists of the contents of the file `README` (or `README.TXT`), followed by a table holding all currently loaded source-files that appear below the given directory (i.e. traversal is *recursive*) and for each file a list of public predicates and their descriptive summary. Finally, if a file `TODO` or `TODO.TXT` exists, its content is added at the end of the directory index.

9 Documentation files

Sometimes it is desirable to document aspects of a package outside the source-files. For this reason the system creates a link to files using the extension `.txt`. The referenced file is processed as Wiki source. The two fragments below illustrate the relation between an `.pl` file and a `.txt` file.

[6] SVG images are included using the `object` element. This is supported by many modern browsers. When using IE, one needs at least IE9.

```
%!      read_setup(+File, -Setup) is det.
%
%       Read application setup information from File.  The details
%       on setup are described in setup.txt.
```

```
---+ Application setup data

If a file =|.myapprc|= exists in the user's home directory the
application will process this data using setup.pl. ...
```

10 Running the documentation system

10.1 During development

To support the developer with an up-to-date version of the documentation of both the application under development and the system libraries the developer can start an HTTP documentation server using the command `doc_server(?Port)`. A good way to deploy PlDoc for program development is to write a file called e.g., `debug.pl` that sets up the preferred development environment and loads your program. below is an example `debug.pl` that starts PlDoc and prints strings as text before loading the remainder of your program.

```
:- doc_server(4000).    % Start PlDoc at port 4000
:- portray_text(true).  % Enable portray of strings

:- [load].              % load your program
```

doc_collect(+*Bool*)
: Enable/disable collecting structured comments into the Prolog database. See `doc_server/1` or `doc_browser/0` for the normal way to deploy the server in your application. All these predicates must be called *before* loading your program.

doc_server(*?Port*)
: Start documentation server at *Port*. Same as `doc_server`(*Port, [allow(localhost), workers(1)]*). This predicate must be called *before* loading the program for which you consult the documentation. It calls `doc_collect/1` to start collecting documentation while (re-)loading your program.

doc_server(*?Port, +Options*)
: Start documentation server at *Port* using *Options*. Provided options are:

  root(+*Path*)
  : Defines the root of all locations served by the HTTP server. Default is `/`. *Path* must be an absolute URL location, starting with `/` and ending in `/`. Intented for public services behind a reverse proxy. See documentation of the HTTP package for details on using reverse proxies.

edit(+*Bool*)
If `false`, do not allow editing, even if the connection comes from localhost. Intended together with the `root` option to make pldoc available from behind a reverse proxy. See the HTTP package for configuring a Prolog server behind an Apache reverse proxy.

allow(+*HostOrIP*)
Allow connections from *HostOrIP*. If *Host* is an atom starting with a '.', suffix matching is preformed. I.e. `allow('.uva.nl')` grants access to all machines in this domain. IP addresses are specified using the library(socket) `ip/4` term. I.e. to allow access from the 10.0.0.X domain, specify `allow(ip(10,0,0,_))`.

deny(+*HostOrIP*)
Deny access from the given location. Matching is equal to the `allow` option.

Access is granted iff

- Both *deny* and *allow* match
- *allow* exists and matches
- *allow* does not exist and *deny* does not match.

doc_browser
Open the user's default browser on the running documentation server. Fails if no server is running.

doc_browser(+*Spec*)
Open the user's default browser on the specified page. *Spec* is handled similar to `edit/1`, resolving anything that relates somehow to the given specification and ask the user to select.[7].

10.2 As a manual server

The library `pldoc/doc_library` defines `doc_load_library/0` to load the entire library.

doc_load_library
Load all library files. This is intended to set up a local documentation server. A typical scenario, making the server available at port 4000 of the hosting machine from all locations in a domain is given below.

```
:- doc_server(4000,
              [ allow('.my.org')
              ]).
:- use_module(library(pldoc/doc_library)).
:- doc_load_library.
```

Example code can be found in `$PLBASE/doc/packages/examples/pldoc`.

10.3 Using the browser interface

The documentation system is normally accessed from a web-browser after starting the server using `doc_server/1`. This section briefly explains the user-interface provided from the browser.

[7] BUG: This flexibility is not yet implemented

10.3.1 Searching

The top-right of the screen provides a search-form. The search string typed is searched as a substring and case-insensitive. Multiple strings seperated by spaces search for the intersection. Searching for objects that do not contain a string is written as `-`⟨*string*⟩. A search for adjacent strings is specified as `"`⟨*string*⟩`"`. Here are some examples:

`load file`	Searches for all objects with the strings `load` and `file`.
`load -file`	Searches for objects with `load`, but *without* `file`.
`"load file"`	Searches for the string `load file`.

The two radio-buttons below the search box can be used to limit the search. All searches both the application and manuals. Searching for Summary also implies Name.

10.3.2 Views

The web-browser supports several views, which we briefly summarise here:

- *Directory*
  In directory-view mode, the contents of a directory holding Prolog source-files is shown file-by-file in a summary-table. In addition, the contents of the `README` and `TODO` files is given.

- *Source File*
  When showing a Prolog source-file it displays the module documentation from the `/** <module ... */` comment and the public predicates with their full documentation. Using the zoom button the user can select to view both public and documentated private predicates. Using the source button, the system shows the source with syntax highlighting as in PceEmacs and formatted structured comments.[8]

- *Predicate*
  When selecting a predicate link the system presents a page with the documentation of the predicate. The navigation bar allows switching to the Source File if the documentation comes from source or the containing section if the documentation comes from a manual.

- *Section*
  Section from the manual. The navigation bars allows viewing the enclosing section (*Up*).

10.3.3 Editing

If the browser is accessed from `localhost`, each object that is related to a known source-location has an edit icon at the right side. Clicking this calls `edit/1` on the object, calling the user's default editor in the file. To use the built-in PceEmacs editor, either set the Prolog flag `editor` to `pce_emacs` or run `?- emacs.` before clicking an edit button.

Prolog source-files have a *reload* button attached. Clicking this reloads the source file if it was modified and refreshes the page. This supports a comfortable edit-view loop to maintain the source-code documentation.

[8] This mode is still incomplete. It would be nice to add line-numbers and links to documentation and definitions in the sources.

10.4 library(doc_files): Create stand-alone documentation files

To be done Generate a predicate index?

Create stand-alone documentation from a bundle of source-files. Typical use of the PlDoc package is to run it as a web-server from the project in progress, providing search and guaranteed consistency with the loaded version. Creating stand-alone files as provided by this file can be useful for printing or distribution.

doc_save(+*FileOrDir, +Options*)
Save documentation for *FileOrDir* to `file(s)`. *Options* include

format(+*Format*)
Currently only supports `html`.

doc_root(+*Dir*)
Save output to the given directory. Default is to save the documentation for a file in the same directory as the file and for a directory in a subdirectory `doc`.

title(+*Title*)
Title is an atom that provides the HTML title of the main (index) page. Only meaningful when generating documentation for a directory.

man_server(+*RootURL*)
Root of a manual server used for references to built-in predicates. Default is `http://www.swi-prolog.org/pldoc/`

index_file(+*Base*)
Filename for directory indices. Default is `index`.

if(*Condition*)
What to do with files in a directory. `loaded` (default) only documents files loaded into the Prolog image. `true` documents all files.

recursive(+*Bool*)
If `true`, recurse into subdirectories.

css(+*Mode*)
If `copy`, copy the CSS file to created directories. Using `inline`, include the CSS file into the created files. Currently, only the default `copy` is supported.

The typical use-case is to document the Prolog files that belong to a project in the current directory. To do this load the Prolog files and run the goal below. This creates a sub-directory `doc` with an index file `index.html`. It replicates the directory structure of the source directory, creating an HTML file for each Prolog file and an index file for each sub-directory. A copy of the required CSS and image resources is copied to the `doc` directory.

```
?- doc_save(., [recursive(true)]).
```

doc_pack(+*Pack*)
Generate stand-alone documentation for the package *Pack*. The documentation is generated

in a directory `doc` inside the pack. The index page consists of the content of `readme` or `readme.txt` in the main directory of the pack and an index of all files and their public predicates.

10.5 Including PlDoc in a LaTeX document

The LaTeX backend aims at producing quality paper documentation as well as integration of predicate description and Wiki files in LaTeX documents such as articles and technical reports. It is realised by the library `doc_latex.pl`.

The best practice for using the LaTeX backend is yet to be established. For now we anticipate processing a Wiki document saved in a .txt file using `doc_latex/3` to produce either a simple complete LaTeX document or a partial document that is included into the the main document using the LaTeX `\input` command. Typically, this is best established by writing a *Prolog Script* that generates the required LaTeX document and call this from a *Makefile*. We give a simple example from PlDoc, creating this section from the wiki-file `latex.txt` below.

```
:- use_module(library(doc_latex)).
:- [my_program].
```

We generate latex.tex from `latex.txt` using this Makefile fragment:

```
.SUFFIXES: .txt .tex

.txt.tex:
        swipl -f script.pl \
            -g "doc_latex('$*.txt','$*.tex',[stand_alone(false)]),halt" \
            -t "halt(1)"
```

10.5.1 Predicate reference for the LaTeX backend

High-level access is provided by `doc_latex/3`, while more low level access is provided by the remaining predicates. Generated LaTeX depends on the style file `pldoc.sty`, which is a plain copy of `pl.sty` from the SWI-Prolog manual sources. The installation installs `pldoc.sty` in the `pldoc` subdirectory of the Prolog manual.

doc_latex(+*Spec*, +*OutFile*, +*Options*) *[det]*

Process one or more objects, writing the LaTeX output to *OutFile*. *Spec* is one of:

Name **/** *Arity*
: Generate documentation for predicate

Name **//** *Arity*
: Generate documentation for DCG rule

File
: If *File* is a prolog file (as defined by `user:prolog_file_type/2`), process using `latex_for_file/3`, otherwise process using `latex_for_wiki_file/3`.

Typically *Spec* is either a list of filenames or a list of predicate indicators. Defined options are:

stand_alone(+*Bool*)
If `true` (default), create a document that can be run through LaTeX. If `false`, produce a document to be included in another LaTeX document.

public_only(+*Bool*)
If `true` (default), only emit documentation for exported predicates.

section_level(+*Level*)
Outermost section level produced. *Level* is the name of a LaTeX section command. Default is `section`.

summary(+*File*)
Write summary declarations to the named *File*.

latex_for_file(+*File, +Out, +Options*) *[det]*
Generate a LaTeX description of all commented predicates in *File*, writing the LaTeX text to the stream *Out*. Supports the options `stand_alone`, `public_only` and `section_level`. See `doc_latex/3` for a description of the options.

latex_for_wiki_file(+*File, +Out, +Options*) *[det]*
Write a LaTeX translation of a Wiki file to the steam *Out*. Supports the options `stand_alone`, `public_only` and `section_level`. See `doc_latex/3` for a description of the options.

latex_for_predicates(+*PI:list, +Out, +Options*) *[det]*
Generate LaTeX for a list of predicate indicators. This does **not** produce the \begin{description}...\end{description} environment, just a plain list of \predicate, etc. statements. The current implementation ignores *Options*.

11 Motivation of choices

Literal programming is an established field. The TEX source is one of the first and best known examples of this approach, where input files are a mixture of TEX and PASCAL source. External tools are used to untangle the common source and process one branch to produce the documentation, while the other is compiled to produce the program.

A program and its documentation consists of various different parts:

- The program text itself. This is the minimum that must be handed to the compiler to create an executable (module).
- Meta information about the program: author, modifications, license, etc.
- Documentation about the overall structure and purpose of the source.
- Description of the interface: public predicates, their types, modes and whether or not they are deterministic as wel as an informative text on each public predicate.
- Description of key private predicates necessary to understand how the public interface is realised.

Structured comments or directives

Comments can be added through Prolog directives, a route taken by Ciao Prolog with lpdoc [Hermenegildo, 2000] and Logtalk [Moura, 2003]. We feel structured comments are a better alternative for the following reasons:

- Prolog programmers are used to writing comments as Prolog comments.
- Using Prolog strings requires unnatural escape sequences for string quotes and long literal values tend to result in hard to find quote-mismatches. Python uses comments in long strings, fixing this problem using a three double quotes to open and close long comments.
- Comments should not look like code, as that makes it more difficult to find the actual code.

We are aware that the above problems can be dealt with using syntax-aware editors. Only a few editors are sufficiently powerful to support this correctly though and we do not expect the required advanced modes to be widely available. Using comments we do not need to force users into using a particular editor.

Wiki or HTML

JavaDoc uses HTML as markup inside the structured comments. Although HTML is more widely known than —for example— LaTeX or TeXinfo, we think the Wiki approach is sufficiently widely known today. Using text with minimal layout conventions taken largely from plaintext newsnet and E-mail, Wiki input is much easier to read in the source-file than HTML without syntax support from an editor.

Types and modes

Types and modes are not a formal part of the Prolog language. Nevertheless, their role goes beyond pure documentation. The test-system can use information about non-determinism to validate that deterministic calls are indeed deterministic. Type information can be used to analyse coverage from the test-suite, to generate runtime type verification or to perform static type-analysis. We have chosen to use a structured comment with formal syntax for the following reasons:

- As a comment, they stay together with the comment block of a predicate. We feel it is best to keep documentation as close as possible to the source.
- As we parse them separately, we can pick up argument names and create a readable syntax without introducing possibly conflicting operators.
- As a comment they do not introduce incompatibilities with other Prolog systems.

Few requirements

SWI-Prolog aims at platform independency. We want tools to rely as much as possible on Prolog itself. Therefore, the entire infrastructure is written in Prolog. Output as HTML is suitable for browsing and not very high quality printing on virtually all platforms. Output to LaTeX requires more infrastructure for processing and allows for producing high-quality PDF documents.

12 Compatibility and standards

Initially, the PlDoc wiki language was based on Twiki. Currently, markdown is a wiki syntax that is widely accepted and not tight to a single system. In PlDoc 2, we have adopted markdown, including many of the limitations and extensions introduced by Docygen. Limitations are needed to avoid ambiguities due to the common use of symbol charaters in programming languages. Extensions are desirable to make use of already existing conventions and support requirements of program documentation.

Some of the changes in PlDoc 2 are to achieve compatibility with the Prolog Commons project. The library documentation conventions of this project will be based on PlDoc and the Ciao lpdoc standards. It is likely that there will be more changes to the PlDoc format to synchronise with Commons. We do not anticipate significant impact on existing documentation.

References

[Covington *et al.*, 2012] Michael A. Covington, Roberto Bagnara, Richard A. O'Keefe, Jan Wielemaker, Simon Price, and Simon Price. Coding guidelines for prolog coding guidelines for prolog. pages 889–927, 2012.

[Hermenegildo, 2000] Manuel V. Hermenegildo. A documentation generator for (c)lp systems. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1345–1361. Springer, 2000.

[Jeffery *et al.*, 2000] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in mercury. In *ACSC*, pages 128–135. IEEE Computer Society, 2000.

[Moura, 2003] Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.

[Mycroft & O'Keefe, 1984] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.

Index