

SWI-Prolog Source Documentation

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: `wielemak@science.uva.nl`

July 31, 2006

Abstract

This article presents PIDoc, the SWI-Prolog source-code documentation infrastructure which is loosely based on JavaDoc.

Contents

1	Introduction	3
2	Overview	3
3	Structured comments	3
4	Type and mode declarations	4
5	Wiki notation	4
5.1	Structuring conventions	5
5.2	Text markup: fonts and links	5
6	Motivation of choices	6
6.1	Wiki or HTML	6

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 measurements need to be adopted. A shared and well designed coding style is one of them. In addition, documentation of source-files and their primary interfaces as well as a testing framework must be established.

Only a few documentation and testing frameworks exist in the Prolog world. In my view they all fall short realising the basic needs in a lightweight and easy to adopt system. We have noticed in various projects as well as through the code we receive in the cause of testing and debugging SWI-Prolog that the discipline to come with consistent style, well commented code and a test-suite is not very well established in the Prolog community. If we want to improve this practice, we should make sure that

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

2 Overview

The PIDoc infrastructure is based on *structured comments*, just like JavaDoc. 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 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 documents.

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 that is seen most in the Prolog community and the block-comment (/...*/) based one commonly seen in the Java and C domain. As we cannot determine the argument-names, type and modes from following (predicate) code itself, we must supply this in the comment.¹ The overall structure of the comment therefore is:

- Semi-formal type- and mode-description
- Wiki-style documentation body
- JavaDoc style keywords (@keyword value)

The Semi-formal type and mode declarations are ended by either a blank line or, using line-comments, by the first line that starts with a single %.

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

¹See section 6.1.

4 Type and mode declarations

The type and mode declaration header consists of one or more Prolog terms. Each term describes a mode of the predicate. The syntax is informally described below:

$\langle \text{modedef} \rangle$	$::=$	$\langle \text{head} \rangle ['/'] \text{'is' } \langle \text{determinism} \rangle$
		$ \langle \text{head} \rangle ['/']$
$\langle \text{determinism} \rangle$	$::=$	'det'
		$ \text{'semidet'}$
		$ \text{'nondet'}$
$\langle \text{head} \rangle$	$::=$	$\langle \text{functor} \rangle (\langle \text{argspec} \rangle \text{' ' } \langle \text{argspec} \rangle)$
	$::=$	$\langle \text{functor} \rangle$
$\langle \text{argspec} \rangle$	$::=$	$[\langle \text{instantiation} \rangle] \langle \text{argname} \rangle [':' \langle \text{type} \rangle]$
$\langle \text{instantiation} \rangle$	$::=$	$\text{'+' } \text{'-'} \text{'?' } \text{'@'} \text{'!'}$
$\langle \text{type} \rangle$	$::=$	$\langle \text{term} \rangle$

Instantiation patterns are:

- + Argument must be fully instantiated to a term that satisfies the type.
- Argument must be unbound.
- ? Argument must be bound to a *partial term* of the indicated type. Note that a variable is a partial term for any type.
- @ Argument is not further instantiated.
- ! Argument contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.

In the current version 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 [Ciao assertion language, O’Keefe and Mycroft, Mercury].

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.
%
%      @iso
```

5 Wiki notation

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

5.1 Structuring conventions

Paragraphs Paragraphs are separated by a blank line.

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:

<code>*</code>	Bulleted list item
<code>1.</code>	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.
<code>\$ Title :</code>	Item Description list item.

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 for a table. Example:

```
| Author      | Jan Wielemaker |
| Copyright   | GPL              |
```

Code (verbatim) Verbatim is embedded between lines containing only `==`, as shown in the example below. The indentation of the `==` must match and the indentation of the verbatim text is reduced by the indentation of the `==` marks.

```
==
small(X) :-
    X < 5.
==
```

5.2 Text markup: fonts and links

Wiki text markup to realise fonts is mostly based on old plaintext conventions in newsgroup and E-mail. We added some Prolog specific conventions to this. For font changing code, The opening symbol must be followed immediately by a word and the closing one must immediately follow a word.

<code>*bold text*</code>	Typset text between ‘*’ in bold
<code>_emphasize_</code>	Typset text between ‘_’ as <code>_emphasized</code>
<code>=code=</code>	Typset text between ‘=’ in fixed font
<code>name/arity</code>	Create a link to a predicate
<code>name//arity</code>	Create a link to a DCG rule
<code>Word</code>	Capitalised words that appear as argument-name are written using <code>\bnfmeta{var}Word</var></code>

6 Motivation of choices

Literal programming is an old field. The \TeX source is one of the oldest and wellknown examples of this approach where input files are a mixture of \TeX and PASCAL source. External tools are used to untangle the common source, 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 well 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 and Logtalk. 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.
- 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.

6.1 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 using text with minimal layout conventions taken largely from plaintext newsgroups and E-mail is more attractive. Wiki input is notably 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 purely 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 predicate 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 to 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 every platform. Output to \LaTeX requires more infrastructure for processing, but allows for producing high-quality PDF documents.

Index

`nb_setarg/3`, 4

`pldoc library`, 3

`setarg/3`, 4