

The lpdoc Documentation Generator

An Automatic Documentation Generator for (C)LP Systems

The CIAO System Documentation Series

Technical Report CLIP 5/97.1

Draft printed on: 27 November 1999

Version 1.9#29 (1999/11/24, 16:47:1 MET)

Manuel Hermenegildo and the CLIP Group

`clip@dia.fi.upm.es`

<http://www.clip.dia.fi.upm.es/>

The CLIP Group

Facultad de Informática

Universidad Politécnica de Madrid

Copyright © 1996-99 Manuel Hermenegildo/The CLIP Group.

This document may be freely read, stored, reproduced, disseminated, translated or quoted by any means and on any medium provided the following conditions are met:

1. Every reader or user of this document acknowledges that is aware that no guarantee is given regarding its contents, on any account, and specifically concerning veracity, accuracy and fitness for any purpose.
2. No modification is made other than cosmetic, change of representation format, translation, correction of obvious syntactic errors, or as permitted by the clauses below.
3. Comments and other additions may be inserted, provided they clearly appear as such; translations or fragments must clearly refer to an original complete version, preferably one that is easily accessed whenever possible.
4. Translations, comments and other additions or modifications must be dated and their author(s) must be identifiable (possibly via an alias).
5. This licence is preserved and applies to the whole document with modifications and additions (except for brief quotes), independently of the representation format.
6. Any reference to the "official version", "original version" or "how to obtain original versions" of the document is preserved verbatim. Any copyright notice in the document is preserved verbatim. Also, the title and author(s) of the original document should be clearly mentioned as such.
7. In the case of translations, verbatim sentences mentioned in (6.) are preserved in the language of the original document accompanied by verbatim translations to the language of the translated document. All translations state clearly that the author is not responsible for the translated work. This license is included, at least in the language in which it is referenced in the original version.
8. Whatever the mode of storage, reproduction or dissemination, anyone able to access a digitized version of this document must be able to make a digitized copy in a format directly usable, and if possible editable, according to accepted, and publicly documented, public standards.
9. Redistributing this document to a third party requires simultaneous redistribution of this licence, without modification, and in particular without any further condition or restriction, expressed or implied, related or not to this redistribution. In particular, in case of inclusion in a database or collection, the owner or the manager of the database or the collection renounces any right related to this inclusion and concerning the possible uses of the document after extraction from the database or the collection, whether alone or in relation with other documents.

Any incompatibility of the above clauses with legal, contractual or judiciary decisions or constraints implies a corresponding limitation of reading, usage, or redistribution rights for this document, verbatim or modified.

Table of Contents

Summary	1
1 Introduction.....	3
1.1 Overview of this document	3
1.2 lpdoc operation - source and target files	3
1.3 lpdoc usage	4
1.4 Version/Change Log (lpdoc)	6
2 Generating Installing and Accessing Manuals..	11
2.1 Generating a manual	11
2.2 Working on a manual	12
2.3 Cleaning up the documentation directory	12
2.4 Installing a generated manual in a public area	13
2.5 Enhancing the documentation being generated	13
2.6 Accessing on-line manuals	14
2.6.1 Accessing html manuals	14
2.6.2 Accessing info manuals	14
2.6.3 Accessing man manuals	15
2.6.4 Putting it all together	15
2.7 Some usage tips	16
2.7.1 Ensuring Compatibility with All Supported Target Formats	16
2.7.2 Writing comments which document version/patch changes	16
2.7.3 Documenting Libraries and/or Applications	17
2.7.4 Documenting files which are not modules	17
2.7.5 Splitting large documents into parts	17
2.7.6 Documenting reexported predicates	17
2.7.7 Separating the documentation from the source file	18
2.7.8 Generating auxiliary files (e.g., READMEs)	18
2.8 Troubleshooting	18
3 Enhancing Documentation with Machine-Readable Comments	21
3.1 Usage and interface (<code>comments</code>)	21
3.2 Documentation on exports (<code>comments</code>)	21
comment/2 (<code>decl</code>)	21
docstring/1 (<code>prop</code>)	21
stringcommand/1 (<code>prop</code>)	22
version_descriptor/1 (<code>regtype</code>)	27
filetype/1 (<code>regtype</code>)	27
3.3 Documentation on internals (<code>comments</code>)	28
version_number/1 (<code>regtype</code>)	28
ymd_date/1 (<code>regtype</code>)	28
time_struct/1 (<code>regtype</code>)	28
version_maintenance_type/1 (<code>regtype</code>)	28

4	Program assertions	31
4.1	More info	31
4.2	Some attention points	31
4.3	Usage and interface (assrt)	32
4.4	Documentation on new declarations (assrt)	32
	pred/1 (decl)	32
	pred/2 (decl)	32
	calls/1 (decl)	33
	calls/2 (decl)	33
	success/1 (decl)	33
	success/2 (decl)	33
	comp/1 (decl)	33
	comp/2 (decl)	34
	prop/1 (decl)	34
	prop/2 (decl)	34
	regtype/1 (decl)	34
	type/1 (decl)	35
	regtype/2 (decl)	35
	type/2 (decl)	35
	entry/1 (decl)	35
	modedef/1 (decl)	36
	decl/1 (decl)	36
	decl/2 (decl)	36
	comment/2 (decl)	37
4.5	Documentation on exports (assrt)	37
	check/1 (pred)	37
	trust/1 (pred)	37
	true/1 (pred)	37
	false/1 (pred)	38
5	Types and properties related to assertions	39
5.1	Usage and interface (assrt_props)	39
5.2	Documentation on exports (assrt_props)	39
	assrt_body/1 (regtype)	39
	head_pattern/1 (prop)	40
	complex_arg_property/1 (regtype)	40
	property_conjunction/1 (regtype)	41
	property_starterm/1 (regtype)	41
	complex_goal_property/1 (regtype)	41
	c_assrt_body/1 (regtype)	42
	s_assrt_body/1 (regtype)	42
	g_assrt_body/1 (regtype)	43
	assrt_status/1 (regtype)	43
	assrt_type/1 (regtype)	44
	predfunctor/1 (regtype)	44
	propfunctor/1 (regtype)	44
	docstring/1 (prop)	44
6	An Example - Documenting a Library Module	
	45

7	Auto Documenter Output for the Example	
	Module	51
7.1	Usage and interface (<code>example_module</code>)	51
7.2	Documentation on exports (<code>example_module</code>)	51
	<code>bar/1 (regtype)</code>	51
	<code>baz/1 (regtype)</code>	51
	<code>aorb/1 (regtype)</code>	52
	<code>tree_of/2 (regtype)</code>	52
	<code>list_or_aorb/2 (regtype)</code>	52
	<code>list/1 (prop)</code>	52
	<code>q/1 (pred)</code>	53
	<code>q/2 (pred)</code>	53
	<code>r/1 (pred)</code>	53
	<code>og/1 (pred)</code>	54
	<code>t/5 (pred)</code>	54
	<code>u/3 (pred)</code>	54
	<code>w/1 (pred)</code>	54
	<code>p/5 (pred)</code>	55
	<code>long/1 (prop)</code>	55
	<code>list/1 (prop)</code>	55
7.3	Documentation on multifiles (<code>example_module</code>)	55
	<code>p/3 (pred)</code>	55
7.4	Documentation on internals (<code>example_module</code>)	57
	<code>s/1 (pred)</code>	57
	<code>list/2 (prop)</code>	57
	<code>og/2 (modedef)</code>	57
	<code>is/2 (pred)</code>	58
8	Installing lpdoc	59
8.1	Installing the source distribution	59
8.2	Installing the binary distribution	59
8.3	Other software packages required	60
	References	61
	Predicate/Method Definition Index	63
	Property Definition Index	65
	Regular Type Definition Index	67
	Mode Definition Index	69
	Concept Definition Index	71
	Global Index	73

Summary

`lpdoc` is an *automatic program documentation generator* for (C)LP systems.

`lpdoc` generates a reference manual automatically from one or more source files for a logic program (including ISO-Prolog, CIAO, many CLP systems, ...). It is particularly useful for documenting library modules, for which it automatically generates a description of the module interface. However, `lpdoc` can also be used quite successfully to document full applications and to generate nicely formatted plain ascii “readme” files. A fundamental advantage of using `lpdoc` to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds.

The quality of the documentation generated can be greatly enhanced by including within the program text:

- *assertions* (types, modes, etc. ...) for the predicates in the program, and
- *machine-readable comments* (in the “*literate programming*” style).

The assertions and comments included in the source file need to be written using the CIAO system *assertion language*. A simple compatibility library is available to make traditional (constraint) logic programming systems ignore these assertions and comments allowing normal treatment of programs documented in this way.

The documentation is currently generated first in `texinfo` format. From the `texinfo` output, printed and on-line manuals in several formats (dvi, ps, info, html, etc.) can be easily generated automatically, using publicly available tools. `lpdoc` can also generate ‘man’ pages (Unix man page format) as well as brief descriptions in html or emacs info formats suitable for inclusion in an on-line index of applications. In particular, `lpdoc` can create and maintain fully automatically WWW and info sites containing on-line versions of the documents it produces.

The `lpdoc` manual (and the CIAO system manuals) are generated by `lpdoc`.

`lpdoc` is distributed under the GNU general public license.

This documentation corresponds to version 1.9#29 (1999/11/24, 16:47:1 MET).

1 Introduction

`lpdoc` is an *automatic program documentation generator* for (C)LP systems.

`lpdoc` generates a reference manual automatically from one or more source files for a logic program (including ISO- `Prolog` [DEDC96], `CIAO` [Bue95], many `CLP` [JM94] systems, ...). It is particularly useful for documenting library modules, for which it automatically generates a description of the module interface. However, `lpdoc` can also be used quite successfully to document full applications and to generate nicely formatted plain ASCII “readme” files. A fundamental advantage of using `lpdoc` to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds.

1.1 Overview of this document

This first part of the document provides basic explanations on how to generate a manual from a set of files that already contain assertions and comments. Examples are given using the files in the `examples` directory provided with the `lpdoc` distribution.

These instructions assume that `lpdoc` (at least the executable and the library) is installed somewhere in your system. Installation instructions can be found in Chapter 8 [Installing `lpdoc`], page 59.

Other parts of this document provide:

- Documentation on the *assertions* that `lpdoc` uses (those defined in the `CIAO assertions` library [PBH97,PBH98,Bue98]).
- Documentation on a basic set of properties, types, etc. to be used in such assertions (those defined in the `CIAO basicprops` library).
- Documentation on the formatting commands that can be embedded in *comments*.

This document is also an internals manual, providing information on how the different internal parts of `lpdoc` are connected, which can be useful if new capabilities need to be added to the system or its libraries are used for other purposes. To this end, the document also provides:

- The documentation for the `autodoc` automatic documentation library, which provides the main functionality of `lpdoc`.
- Documentation on the predicates that define the conversion formats used (`texinfo`, and others), and which are in the `autodocformats` library.

All of the above have been generated automatically from the assertions in the corresponding sources and can also be seen as examples of the use of `lpdoc`.

1.2 `lpdoc` operation - source and target files

The main input used by `lpdoc` in order to generate a manual are Prolog source files. Basically, `lpdoc` generates a file in the GNU `texinfo` format (with a `.texi` ending) for each `Prolog` file (see “The GNU Texinfo Documentation System” manual for more info on this format). The `Prolog` files must have a `.pl` ending.

If the `.pl` file does not define the predicates `main/0` or `main/1`, it is assumed to be a *library* and it is documented as such: the `.texi` file generated will contain information on the interface (e.g., the predicates exported by the file, the name of the module and usage if it is a module, etc.), in addition to any other machine readable comments included in the file (see Section 2.5 [Enhancing the documentation being generated], page 13). If, on the contrary, the file defines the predicates `main/0` or `main/1`, it is assumed to be an *application* and no description of the interface is generated (see Section 2.7 [Some usage tips], page 16).

If needed, files written directly in `texinfo` can also be used as input files for `lpdoc`. These files *must have a .src (instead of .texi) ending*. This is needed to distinguish them from any automatically generated `.texi` files. Writing files directly in `texinfo` has the disadvantage that it may be difficult to adhere to all the conventions used by `lpdoc`. For example, these files will be typically used as chapters and must be written as such. Also, the set of indices used must be the same that `lpdoc` is generating automatically. Finally, no bibliographic citations can be used. Because of this, and because in the future `lpdoc` may be able to generate documentation in formats other than `texinfo` directly (in which case these files would not be useful), writing files in `texinfo` directly is discouraged. This facility was added mainly to be able to reuse parts of manuals which were already written in `texinfo`. Note that if a stand-alone file needs to be written (i.e., a piece of documentation that is not associated to any `.pl` file) it can always be written as a “dummy” `.pl` file (i.e., one that is not used as code), but which contains machine readable comments).

A manual can be generated either from a single source file (`.pl` or `.src`) or from a set of source files. In the latter case, then one of these files should be chosen to be the *main file*, and the others will be the *component files*. The main file is the one that will provide the title, author, date, summary, etc. to the entire document. In principle, any set of source files can be documented, even if they contain no assertions or comments. However, the presence of these will greatly improve the documentation (see Section 2.5 [Enhancing the documentation being generated], page 13).

If the manual is generated from a single main file (i.e., `COMPONENTS`, defined below, is empty), then the document generated will be a flat document containing no chapters. If the manual is generated from a main file and one or more components, then the document will contain chapters. The comments in the main file will be used to generate the introduction, while each of the component files will be used to generate a separate chapter. The contents of each chapter will be controlled by the contents of the corresponding component file.

As mentioned before, `lpdoc` typically generates `texinfo` files. From the `texinfo` files, `lpdoc` can generate printed and on-line manuals in several formats (`dvi`, `ps`, `ascii`, `html`, `info`, etc.) automatically, using different (publicly available) packages. Documentation in some other formats (e.g., `man1` pages) can be generated directly by `lpdoc`, selecting the appropriate options (see below). `lpdoc` can also generate directly includes generating (parts of) a master index of documents which can be placed in an installation directory and which will provide pointers to the individual manuals generated. Using this feature, `lpdoc` can maintain global `html` and/or `info` documentation sites automatically (see Section 2.4 [Installing a generated manual in a public area], page 13).

1.3 lpdod usage

The following provides the different command line options available when invoking `lpdoc`. This description is intended only for advanced users which might like to use `lpdoc` in custom applications. Note that the normal way to use `lpdoc` is by setting parameters in a `SETTINGS` file (see Section 2.1 [Generating a manual], page 11), and `lpdoc` will be invoked automatically with the correct options by the `Makefile` provided with the distribution.

```
lpdoc -help
```

Print this help message.

```
lpdoc [MiscOpts] [-l <path1> ... <pathN> ]
      [-s <path1> ... <pathN> ]
      [-i <i1> ... <iN> ]
      [-u path_aliases_file ]
```

```
-main <main> [ <f1.texic> ... <fN.texic> ]
```

Generate the main .texic file for main application file <file> and whose component .texic files are <file1.texic>... <fileN.texic>. The optional arguments preceded by -l are the directory paths where files used (via use_module/1) by the file being documented (main in this case) can be found (-s is similar, but putting paths in the -s list flags that these paths are 'system' paths). They are treated in the same way as 'library paths'. The optional arguments preceded by -i are the names of the indices which should be generated. The special index name 'all' results in all indices being generated.

```
lpdoc [MiscOpts] [-l <path1> ... <pathN> ]
                    [-s <path1> ... <pathN> ]
                    [-i <idx1> ... <idxN> ]
                    [-u path_aliases_file ]
                    -component <file.pl>
```

Generate a .texic file for module <file.pl>. The optional arguments are as above.

```
lpdoc [MiscOpts] [-l <path1> ... <pathN> ]
                    [-s <path1> ... <pathN> ]
                    [-u path_aliases_file ]
                    -htmlindex <main>
```

Generate (part of an) html file suitable for including in an html page.

```
lpdoc [MiscOpts] [-l <path1> ... <pathN> ]
                    [-s <path1> ... <pathN> ]
                    [-u path_aliases_file ]
                    -man <main>
```

Generate a man page.

```
lpdoc [MiscOpts] [-l <path1> ... <pathN> ]
                    [-s <path1> ... <pathN> ]
                    [-u path_aliases_file ]
                    -infoindex <main>
```

Generate an info directory entry suitable for including in an info directory.

1.4 Version/Change Log (lpdoc)

Version 1.9 (1999/7/8, 18:19:43 MEST)

In this release the name of the application has changed to **lpdoc** which was found more appropriate, since several formats are generated in addition to **texi**. The major changes are listed below.

New commands:

- `@begin{cartouche}` and `@end{cartouche}` commands now supported.
- `@footnote` command now supported.
- New `gmake htmlview` command (makes a running **netscape** visit the generated html manual). Suggested by Per Cederberg.
- New `gmake distclean` command, intended for software distributions. Leaves the generated documents and eliminates *all* intermediate files (including `.texic/.texi` files).
- Adobe **pdf** format now supported as a valid target. Unfortunately, embedded **.eps** figures are not supported at this time in pdf output.
- The second argument of `:- comment(hide,...)` and `:- comment(doinclude,...)` declarations can now be a list of predicate names.
- A `-u File` option is now supported so that a file including, e.g., path alias definitions can be included (this has the same functionality as the `-u` option in **ciaoc**).
- Now typing just `gmake` does nothing. In order to do something at least one target should be specified. This was necessary so that recursive invocations with empty arguments did nothing.
- Added a new filetype: **part**. This allows splitting large documents into parts, each of which groups a series of chapters.

Other new functionality:

- A style sheet can now be specified which allows modifying many characteristics of the html output (fonts, colors, background, ...) (thanks to Per Cederberg).
- Added limited support for changing page numbering (in **SETTINGS** file).
- The concept indexing commands (`@index`, `@cindex`, and `@concept`) now work somewhat differently, to make them consistent with other indexing commands.
- The old *usage* index is now called, more appropriately, *global* index. Correspondingly, changed things so that now every definition goes to the global index in addition to its definitional index.
- Imported files from module **user** are now documented separately.
- Now a warning is issued if characters unsupported by info are used in section names.
- Navigation in html docs was improved.
- The table of contents in printed manuals now contains entries for the individual descriptions of predicates, props, regtypes, declarations, etc. This can be shut off with the `-shorttoc` option.
- Made more silent in normal conditions: file inclusion is muted now unless `-v` option is selected.
- A single `.texi` file is now constructed (by grouping the `.texic` files generated for all components) in which the references and menus are resolved. This has the advantage that the process of resolving references and menus has now been sped up very significantly. Also, **texi** is now a valid target (perhaps useful for distributions). The generated files now have **texic** (*texinfo component*).

- Now, declarations are always documented as long as there is a `decl` assertion. Also, they are now documented in a separate section.

Bug fixes and other minor improvements:

- The directory containing html manual is now called *BASENAME_html* instead of just *BASENAME*, which was confusing.
- Now requesting building a .ps only does not leave a .dvi behind (useful for distributions).
- File names can now include the symbol `_` even if they contain figures.
- TeX-related intermediate files are now cleaned up after each run in order to avoid clutter.
- Fixed `-modes`, which was broken since going to the new normalizer (was normalizer problem). Fixed problem with no documentation when only modes given.
- Fixed duplication of documentation for internal predicates when also exported.
- Minor formatting problem when no documentation nor definition found for a regtype fixed.
- Determining exports, imports, etc. now done solely by calls to `c_itf` library (and, thus, synchronized with `ciao` compiler).

(Manuel Hermenegildo)

Version 1.8 (1999/3/24, 21:15:33 MET)

This version completes the port to using the `ciao` 0.8 modular assertion processing library. In addition, it includes the following improvements:

- Now, if the name of a file being documented ends in `_doc`, the `_doc` part is left out when referring to the file in the documentation (useful if one would like to place the documentation declarations in different file).
- It is now possible to declare (via a `comment/2` declaration) the intended use of a file which is not a module (i.e. a package, user, or include file), which results in correct documentation of operator definitions, new declarations, etc. The declaration is only needed for 'user' files (i.e., files to be loaded with `ensure_loaded/1`).
- Separated generation of the manuals from their installation. I.e., `gmake install` now does not force a `gmake all`, which has to be done by hand. This was necessary to ensure correct installation of distributed manuals, even if modification dates are changed during installation. Previously, in some cases generation was triggered unnecessarily.
- New `-v` option allows using quieter by default operation when not debugging.
- New option `-propmods` makes the name of the module in which a property is defined appear in front of the property in the places where the property is used.
- New option `-noisoline` makes the textual explanation of the `iso/1` property not appear in the description of the usage (but the `ISO` symbol does appear).
- Two new options, `'-nosysmods'` and `'-noengmods'`, selectively avoid listing the system or engine libraries used.
- If there is no declaration for a predicate, now a line is output with the name and arity and a simple comment saying that there is no further documentation available (this has the great advantage that then it goes in the index, and, for example in `ciao`, they get added to completion commands!).
- Now, if a property or regtype declaration has no textual comment, the actual definition is given (first level only) in the place where it is documented, and a simple generic message where it is used.

- Added `@noindent` and `@iso` commands.
- Nicier spacing now when printing predicate names which are operators, as well as modes, etc.
- Reporting of versions in libraries has been improved: now both the global version and the last version in which the library itself was changed are reported.
- Exported new declarations also documented now for include-type files.
- A module is now documented even if exports nothing at all.
- Engine modules used now documented even if no other modules used (was a reported bug).
- Fixed indexing of names containing `@` etc. for newer versions of texinfo.
- Tabs in verbatim modes now converted to a number of spaces (8). Not perfect, but produces better output than leaving the tabs in.
- Tex is now run in 'nonstopmode' which means it will typically not stop if there are minor errors (but some errors may go unnoticed...).
- The full path of the version maintenance directory is now computed (correctly) using the directory of the `.pl` file being documented as base.
- Notices for missing subtitle, copyright, and summary now only given from main file and not for components.
- Added special handling of `regtype` and generalized it to handle some props specially if there is a certain `comp` property present.

(Manuel Hermenegildo)

Version 1.7 (1998/12/2, 17:43:50 MET)

Major port to use the ciao 0.8 modular assertion processing library. (Manuel Hermenegildo)

Version 1.6 (1998/9/8, 12:49:26 MEST)

Added support for inserting images (.eps files) in text via `@image` command, email addresses via `@email` command, and url references via `@uref` command.

Unix 'man' output much improved. Also, it now includes a usage section. The corresponding text must be given in a string contained in the first argument of a fact of the `usage_message/1` predicate which appears in the program. Also, formatting of 'man' pages has been greatly improved.

A new 'ascii' format is now supported: a simple minded ascii manual (basically, an info file without pointers).

(Manuel Hermenegildo)

Version 1.5 (1998/8/23, 20:30:32 EST)

Now supporting a `@cite` command (YES!). It automatically accesses the bib entries in `.bib` files (using `bibtex`) and produces a 'References' appendix. `@cite` can be used in the text strings exactly as ite in LaTeX. The set of bib files to be used is given in the `SETTINGS` file.

Defining the type of version maintenance that should be performed by the `emacs ciao.el` mode (i.e., whether version numbers are in a given directory or in the file itself) is controlled now via a standard `comment/2` declaration. You should now write a declaration such as:

```
:- comment(version_maintenance,dir('../version')).
```

to state that control info is kept in directory `../version`. This has the advantage that it is shorter than the previous solution and that lpd doc can read this info easily. Using this guarantees that the version numbers of the manuals always coincide with those of the software.

Generation of indices of manuals (.htmlbullet files): if several manuals are installed in the same directory, an index to them is now generated at the beginning of the html cover page describing the directory.

(Manuel Hermenegildo)

Version 1.4 (1998/8/4, 19:10:35 MET DST)

The set of paths defined in **SETTINGS** for finding the source files are now also used to find 'included' files. As a result, full path is not needed any more in, e.g, **@include** command.

New **@ref** command which can be used to refer to chapter, sections, subsections, etc..

Support for recent minor changes in assertion format, including '#' as comment separator.

Used modules are now separated in documentation (in the interface description) by type (user, system, engine...).

Supports new 'hide' option in comments, to prevent an exported predicate from being documented. This is useful for example for avoiding mentioning in the documentation multifile predicates which are not intended to be modified by the user.

(Manuel Hermenegildo)

Version 1.3 (1998/7/10, 16:35:2 MET DST)

Exports are now listed in the chapter header separated by kind (pred, types, properties, ...).

The list of other modules used by a module is now separated in the chapter header into User and System modules (controlled by two sets of paths in **SETTINGS**).

New *hide* option of comment/2 decl prevents an exported predicate from being included in the documentation: `:- comment(hide,p/3)`.

(Manuel Hermenegildo)

Version 1.2 (1998/6/4, 9:12:19 MET DST)

Major overall improvements... (Manuel Hermenegildo)

Version 1.1 (1998/3/31)

Incorporated autodoc and autodoformats library to source in order to make distribution standalone. Improvements to installation and documentation. **Makefiles** now also install documentation in public areas and produce global indices. Several documents can coexist in the same installation directory. (Manuel Hermenegildo)

Version 1.0 (1998/2/24)

First CIAO-native distribution, with installation. (Manuel Hermenegildo)

Version 0.9 (1998/2/24)

Intermediate version, preparing for first major release. Modified **Makefile** and **SETTINGS** to handle installation of manuals. (Manuel Hermenegildo)

Version 0.6 (1998/2/10)

Added new indices and options, as well as more orthogonal handling of files. (Manuel Hermenegildo)

Version 0.4 (1998/2/24)

Added support for **nroff -m** formatting (e.g., for man pages). Added support for optional selection of indices to be generated. Added support for reexported predicates. Added (low level) **ascii** format. Added option handling (**-nbugs -noauthors -noverion -nochangelog -nopatches -modes** and **-headprops ...**). **-literalprops**. Fixed presentation when there are multiple kinds of assertions. Better error checking for **includefact/includedef**. (Manuel Hermenegildo)

Version 0.3 (1998/2/10)

Changed file reader to use CIAO native builtins. As a result, syntax files and full CIAO syntax now supported. Major reorganization of the code to make formatting more orthogonal. Now applications and libraries can be components or main files, standalone or with components interchangeably. `@includefact`, new predicate types, used libraries now precisely detected, `docinclude` option. (Manuel Hermenegildo)

Version 0.2 (1997/12/16)

Ported to native ciao. Version handling, selection of indices, `@include`. Added generation of an html brief description for a global index. Added unix manual page generation. Added support for specifying library paths. `-l` option for `htmlindex` and `man`. Installation improved: now all files for one application in the same directory. (Manuel Hermenegildo)

Version 0.1 (1997/7/30)

First official version (major rewrite from several previous prototypes, autodocumented!). (Manuel Hermenegildo)

Version 0.0 (1996/10/10)

First prototype.

2 Generating Installing and Accessing Manuals

This part describes how to generate a manual (semi-)automatically from a set of source files using `lpdoc`, how to install it in a public area, and how to access it on line. It also includes some recommendations for improving the layout of manuals, usage tips, and troubleshooting advice.

2.1 Generating a manual

The `lpdoc` library directory includes generic *Makefile* and *SETTINGS* files which are quite useful for the actual generation of complete manuals. Use of these files is strongly recommended. Generating a manual using these files involves the following steps:

- Create a directory (e.g., `doc`) in which the documentation will be built. It is highly recommended that this be a separate and initially empty directory. This directory is typically created in the top directory of the distribution of the application or library to be documented.
- Copy the file `SETTINGS` and copy (or, much better, link, since you will typically will not need to change it) the file `Makefile` from the `lpdoc` library directory into the `doc` directory just created. The location of the `lpdoc` library directory is installation-dependent (see Section 8.1 [Installing the source distribution], page 59).
- Edit `SETTINGS` to suit your needs. It is required that you set the following:
 - Set the variable `FILEPATHS` to include all the directories where the files to be documented can be found.
 - Set the variable `SYSTEMPATHS` to include all the *system* directories where system files used (whether they are to be documented or not) can be found.

It is very important to include *all* related directories either in `FILEPATHS` or in `SYSTEMPATHS` because on startup `lpdoc` has *no default search paths for files* defined (not even those typically defined by default in the *Prolog* system under which it was compiled! – this allows documenting *Prolog* systems other than that under which `lpdoc` was compiled).

The effect of putting a path in `SYSTEMPATHS` instead of in `FILEPATHS` is that the modules and files in those paths are documented as *system modules* (this is useful when documenting an application to distinguish its parts from those which are in the system libraries).

- Set `MAIN` to be the name of the *main file* of the application or library (e.g., `main.pl`, `header.src`, etc.).
- Set `COMPONENTS` to a (possibly empty) list, separated by spaces, of source files. These are the *component files*.

For the rest of the settings in the `SETTINGS` file you can simply use the default values indicated. You may however want to change several of these:

- `MAINOPTS` can be set to a series of options which allow more detailed control of what is included in the documentation for the main file and how (i.e., inclusion of bug information, versions and patches or only patches, authors, changelog, explanation of modes, etc.). See `option_comment/2` in `autodocformats` or type `lpdoc -help` for a list of these options.
- In the same way `COMPOPTS` sets options for the component files. Currently these options are common to all component files but they can be different from `MAINOPTS`. The allowable options are the same as above.
- `DOCFORMATS` determines the set of formats (`dvi`, `ps`, `ascii`, `html`, `info`, `man1`, ...) in which the documentation should be generated by default when typing `gmake all`. Selecting `htmlindex` and/or `infoindex` requests the generation of (parts of) a master

index to be placed in an installation directory and which provide pointers to the documents generated (see below). If the main file is an **application**, and the **man1** option is selected, then **lpdoc** looks for a **usage_message/1** fact, which should contain a string as argument, and will use that string to document the *usage of the application* (i.e., it will be used to fill in the *synopsis section of the man page*).

- **INDICES** determines the list of indices to be included at the end of the document. These can include indices for defined predicates, modules, concepts, etc. For a complete list of the types of indices available see **index_comment/2** in **autodocformats** or type **lpdoc -help** for a listing. A setting of **all** generates all the supported indices – but *beware of limitations in the number of simultaneous indices* supported in many **texinfo** installations.

- **BIBFILES** determines a list (separated by commas, full paths, no spaces) of *.bib files*, i.e., files containing *bibliographic entries* in **bibtex** format. This is only relevant if you are using citations in the text (using the **@cite** command). In that case those will be the files in which the citations will be searched for. All the references will appear together in a *References* appendix at the end of the manual.

If you are not using citations, then select the **-norefs** option on the main file, which will prevent an empty 'References' appendix from appearing in the manual.

- **STARTPAGE** (default value 1) allows changing the page number of the first page of the manual. This can be useful if the manual is to be included in a larger document or set of manuals. Typically, this should be an *odd* number.
- Type **gmake all** (you can also type **gmake dvi**, **gmake html**, **gmake ps** or **gmake info**, ... to force generation of a particular target).

2.2 Working on a manual

In order to speed up processing while developing a manual, it is recommended to work by first generating a *.dvi* version only (i.e., by typing **gmake dvi**). The resulting output can be easily viewed by tools such as **xdvi** (which can be started by simply typing **gmake view**). Note that once an **xdvi** window is started, it is not necessary to restart it every time the document is reformatted (**gmake dvi**), since **xdvi** automatically updates its view every time the *.dvi* file changes. This can also be forced by typing **Ⓡ** in the **xdvi** window. The other formats can be generated later, once the *.dvi* version has the desired contents.

2.3 Cleaning up the documentation directory

Several cleanup procedures are provided by the **Makefile**:

- **gmake clean** deletes all intermediate files, but leaves the targets (i.e., the *.ps*, *.dvi*, *.ascii*, *.html*, etc. files), as well as all the generated *.texic* files.
- **gmake distclean** deletes all intermediate files and the generated *.texic* files, leaving only the targets (i.e., the *.ps*, *.dvi*, *.ascii*, *.html*, etc. files). This is the option normally used when building software distributions in which the manuals come ready made in the distribution itself and will not need to be generated during installation.
- **gmake realclean** deletes all intermediate files and the generated targets, but leaves the *.texic* files. This option can be used in software distributions in which the manuals in the different formats will be generated during installation. This is generally more compact, but requires the presence of several tools, such as **tex**, **emacs**, etc. (see Section 8.3 [Other software packages required], page 60), in order to generate the manuals in the target formats during installation.
- **gmake braveclean** performs a complete cleanup, deleting also the *.texic* files, i.e., it typically leaves only the **SETTINGS** and **Makefile**. This is the most compact, but requires in order

to generate the manuals in the target formats during installation, the presence of the tools mentioned above, `lpdoc`, and the source files from which the manuals are generated.

2.4 Installing a generated manual in a public area

Once the manual has been generated in the desired formats, the `Makefile` provided also allows automatic installation in a different area, specified by the `DOCDIR` option in the `SETTINGS` file. This is done by typing `gmake install`.

As mentioned above, `lpdoc` can generate directly brief descriptions in `html` or `emacs` info formats suitable for inclusion in an on-line index of applications. In particular, if the `htmlindex` and/or `infoindex` options are selected, `gmake install` will create the installation directory, place the documentation in the desired formats in this directory, and produce and place in the same directory suitable `index.html` and/or `dir` files. These files will contain some basic info on the manual (extracted from the summary and title, respectively) and include pointers to the relevant documents which have been installed. The variables `HTMLINDEXHEADFILE` / `HTMLINDEXTAILFILE` and `INFODIRHEADFILE` / `INFODIRTAILFILE` (default examples, used in the CLIP group at UPM, are included with the distribution) should point to files which will be used as head and tail templates when generating the `index.html` and/or `dir` files.

Several manuals, coming from different `doc` directories, can be installed in the same `DOCDIR` directory. In this case, the descriptions of and pointers to the different manuals will be automatically combined (appearing in alphabetic order) in the `index.html` and/or `dir` indices, and a *contents area* will appear at the beginning of the *html index page*. If only one manual is installed, selecting the `-nobullet` option for the main file prevents the bullet item from appearing in this contents area.

Note that, depending on the structure of the manuals being generated, some formats are not very suitable for public installation. For example, the `.dvi` format has the disadvantage that it is not self contained if images are included in the manual.

The `Makefile` also makes provisions for manual deinstallation from the installation area. Typing `gmake uninstall` in a `doc` directory will deinstall from `DOCDIR` the manuals corresponding to the `Makefile` in that `doc` directory. If a manual is already installed and changes in the number of formats being installed are desired, `gmake uninstall` should be made before changing the `DOCFORMATS` variable and doing `gmake install` again. This is needed in order to ensure that a complete cleanup is performed.

2.5 Enhancing the documentation being generated

The quality of the documentation generated can be greatly enhanced by including within the program text:

- *assertions*, and
- *machine-readable comments*.

Assertions are declarations which are included in the source program and provide the compiler with information regarding characteristics of the program. Typical assertions include type declarations, modes, general properties (such as *does not fail*), standard compiler directives (such as `dynamic/1`, `op/3`, `meta_predicate/1...`), etc. When documenting a module, `lpdoc` will use the assertions associated with the module interface to construct a textual description of this interface. In principle, only the exported predicates are documented, although any predicate can be included in the documentation by explicitly requesting it (see the documentation for the `comment/2` declaration). Judicious use of these assertions allows at the same time documenting the program code, documenting the external use of the module, and greatly improving the debugging process. The latter is possible because the assertions provide the compiler with information on the intended meaning or behaviour of the program (i.e., the specification) which

can be checked at compile-time (by a suitable preprocessor/static analyzer) and/or at run-time (via checks inserted by a preprocessor).

Machine-readable comments are also declarations included in the source program but which contain additional information intended to be read by humans (i.e., this is an instantiation of the *literate programming* style of Knuth [Knu84]). Typical comments include title, author(s), bugs, changelog, etc. Judicious use of these comments allows enhancing at the same time the documentation of the program text and the manuals generated from it.

lpdoc requires these assertions and comments to be written using the CIAO system *assertion language*. A simple compatibility library is available in order to make it possible to compile programs documented using assertions and comments in traditional (constraint) logic programming systems which lack native support for them (see the `compatibility` directory in the lpdod library). Using this library, such assertions and comments are simply ignored by the compiler. This compatibility library also allows compiling lpdod itself under (C)LP systems other than the CIAO system under which it is developed.

2.6 Accessing on-line manuals

As mentioned previously, it is possible to generate on-line manuals automatically from the `.texic` files, essentially `.html`, `.info`, and `man` files. This is done by simply including the corresponding options in the list of `DOCFORMATS` in the `SETTINGS` file and typing `gmake all`. We now address the issue of how the different manuals can be read on-line.

2.6.1 Accessing html manuals

Once generated, the `.html` files can be viewed using any standard WWW browser, e.g., `netscape` (a command `gmake htmlview` is available which, if there is an instance of `netscape` running in the machine, will make that instance visit the manual in `html` format). To make these files publically readable on the WWW, they should be copied into a directory visible by browsers running in other machines, such as `/home/clip/public_html/lpdod_docs`, `/usr/home/httpd/html/docs/lpdod_docs`, etc. As mentioned before, this is easily done by setting the `DOCDIR` variable in the `SETTINGS` file to this directory and typing `gmake install`.

2.6.2 Accessing info manuals

Generated `.info` files are meant to be viewed by the `emacs` editor or by the standalone `info` application, both publically available from the GNU project sites. To view the a generated `info` file from `emacs` manually (i.e., before it is installed in a common area), type `C-u M-x info`. This will prompt for an `info` file name. Input the name of the `info` file generated by lpdod (`MAIN.info`) and `emacs` will open the manual in `info` mode.

There are several possibilities in order to install an `.info` file so that it is publically available, i.e., so that it appears automatically with all other `info` manuals when starting `info` or typing `C-u M-x info` in `emacs`:

- **Installation in the common info directory:**

- Move the `.info` file to the common `info` directory (typically `/usr/info`, `/usr/local/info`, ..). This can be done automatically by setting the `DOCDIR` variable in the `SETTINGS` file to this directory and typing `gmake install`.

Warning: if you are installing in an `info` directory that is not maintained automatically by lpdod, make sure that you have not selected the `infoindex` option in `DOCFORMATS`, since this will overwrite the existing `dir` file).

- Add an entry to the `info` index in that directory (normally a file in that directory called `dir`). The manual should appear as part of the normal set of manuals available when typing `M-x info` in `emacs` or `info` in a shell. See the `emacs` manual for details.

- **Installation in a different info directory:** you may want to place one or more manuals generated by `lpdoc` in their own directory. This has the advantage that `lpdoc` will maintain automatically an index for all the `lpdoc` generated manuals installed in that directory. In order for such manuals to appear when typing `M-x info` in `emacs` or `info` in a shell there are two requirements:

- This directory must contain a `dir` index. The first part of the process can all be done automatically by setting the `DOCDIR` variable in the `SETTINGS` file to this directory, including the `infoindex` option in `DOCFORMATS`, and typing `gmake install`. This will install the `info` manual in directory `DOCDIR` and update the `dir` file there. `gmake uninstall` does the opposite, eliminating also the manual from the index.
- The directory must be added to the *info path list*. The easiest way to do this is to set the `INFOPATH` environment variable. For example, assuming that we are installing the `info` manual in `/home/clip/public_html/lpdoc_docs` and that `/usr/info` is the common `info` directory, for `csh` in `.cshrc`:

```
setenv INFOPATH /usr/info:/home/clip/public_html/lpdoc_docs
```

Adding the directory to the `info` path list can also be done within `emacs`, by including the following line in the `.emacs` file:

```
(defun add-info-path (newpath)
  (setq Info-default-directory-list
    (cons (expand-file-name newpath) Info-default-directory-list)))
(add-info-path "/home/clip/public_html/lpdoc_docs")
(add-info-path "/usr/info/")
```

However, this has the disadvantage that it will not be seen by the standalone `info` command.

Automatic, direct on-line access to the information contained in the `info` file (e.g., going automatically to predicate descriptions by clicking on predicate names in programs in an `emacs` buffer) can be easily implemented via existing `.el` packages such as `word-help`, written by Jens T. Berger Thielemann (jensthi@ifi.uio.no). `word-help` may already be in your `emacs` distribution, but for convenience the file `word-help.el` and a `word-help-setup.el` file, providing suitable initialization are included in the `lpdoc` library. A suitable interface for `word-help` is also provided by the `ciao.el` `emacs` file that comes with the `CIAO` system distribution (i.e., if `ciao.el` is loaded it is not necessary to load or initialize `word-help`).

2.6.3 Accessing man manuals

The `unix man` format manuals generated by `lpdoc` can be viewed using the `unix man` command. In order for `man` to be able to locate the manuals, they should be copied to one of the subdirectories (e.g., `/usr/local/man/man1`) of one of the main `man` directories (in the previous case the main directory would be `/usr/local/man`). As usual, any directory can be used as a `man` main directory, provided it is included in the environment variable `MANPATH`. Again, this process can be performed automatically by setting the `DOCDIR` variable in the `SETTINGS` file to this directory and typing `gmake install`.

2.6.4 Putting it all together

A simple, powerful, and very convenient way to use the facilities provided by `lpdoc` for automatic installation of manuals in different formats is to install all manuals in all formats in the same directory `DOCDIR`, and to choose a directory which is also accessible via `WWW`. After setting `DOCDIR` to this directory in the `SETTINGS` file, and selecting `infoindex` and `htmlindex` for the `DOCFORMATS` variable, `gmake install/gmake uninstall` will install/uninstall all manuals in all the selected formats in this directory and create and maintain the corresponding `html` and `info` indices. Then, setting the environment variables as follows (e.g., for `csh` in `.cshrc`):

```
setenv DOCDIR    /home/clip/public_html/lpdod_docs
setenv INFOPATH  /usr/local/info:${DOCDIR}
setenv MANPATH   ${DOCDIR}:${MANPATH}
```

Example files for inclusion in user's or common shell initialization files are included in the lpdod library.

More complex setups can be accomodated, as, for example, installing different types of manuals in different directories. However, this currently requires changing the `DOCFORMATS` and `DOCDIR` variables and performing `gmake install` for each installation format/directory.

2.7 Some usage tips

This section contains additional suggestions on the use of lpdod.

2.7.1 Ensuring Compatibility with All Supported Target Formats

One of the nice things about lpdod is that it allows generating manuals in several formats which are quite different in nature. Because these formats each have widely different requirements it is sometimes a little tricky to get things to work successfully for all formats. The following recommendations are intended to help in achieving useful manuals in all formats:

- The best results are obtained when documenting code organized as a series of libraries, and with a well-designed module structure.
- `texinfo` supports only a limited number of indices. Thus, if you select too many indices in the `SETTINGS` file you may exceed `texinfo`'s capacity (which it will signal by saying something like "No room for a new @write").
- The GNU info format requires all *nodes* (chapters, sections, etc.) to have different names. This is ensured by lpdod for the automatically generated sections (by appending the module or file name to all section headings). However, care must be taken when writing section names manually to make them different. For example, use "lpdod usage" instead of simply "Usage", which is much more likely to be used as a section name in another file being documented.
- Also due to a limitation of the `info` format, do not use `:` or `,` or `--` in section, chapter, etc. headings.
- The character `"_"` in names may sometimes give problems in indices, since current versions of `texinfo` do not always handle it correctly.

2.7.2 Writing comments which document version/patch changes

When writing version comments (`:- comment(version(...), "...").`), it is useful to keep in mind that the text can often be used to include in the manual a list of improvements made to the software since the last time that it was distributed. For this to work well, the textual comments should describe the significance of the work done for the user. For example, it is more useful to write "added support for `pred` assertions" than "modifying file so `pred` case is also handled".

Sometimes one would like to write version comments which are internal, i.e., not meant to appear in the manual. This can easily be done with standard Prolog comments (which lpdod will not read). An alternative and quite useful solution is to put such internal comments in *patch* changes (e.g., 1.1#2 to 1.1#3), and put the more general comments, which describe major changes to the user and should appear in the manual, in *version* changes (e.g., 1.1#2 to 1.2#0). Selecting the appropriate options in lpdod then allows including in the manual the version changes but not the patch changes (which might on the other hand be included in an *internals manual*).

2.7.3 Documenting Libraries and/or Applications

As mentioned before, for each a `.pl` file, `lpdoc` tries to determine whether it is a library or the main file of an application, and documents it accordingly. Any combination of libraries and/or main files of applications can be used arbitrarily as components or main files of a `lpdoc` manual. Some typical combinations are:

- *Main file is a library, no components:* A manual of a simple library, which appears externally as a single module. The manual describes the purpose of the library and its interface.
- *Main file is an application, no components:* A manual of a simple application.
- *Main file is a library, components are also libraries:* This can be used for example for generating an internals manual of a library. The main file describes the purpose and use of the library, while the components describe the internal modules of the library.
- *Main file is an application, components are libraries:* This can be used similarly for generating an internals manual of an application. The main file describes the purpose and use of the application, while the components describe the internal modules which compose the application.
- *Main file is a (pseudo-)application, components are libraries:* A manual of a complex library made up of smaller libraries (for example, the `Prolog` library). The (pseudo-)application file contains the introductory material (title, version, etc.). Each chapter describes a particular library.
- *Main file is a (pseudo-)application, components are applications:* This can be used to generate a manual of a set of applications (e.g., a set of utilities). The (pseudo-)application file contains the introductory material (title, version, etc.). Each chapter describes a particular component application.

2.7.4 Documenting files which are not modules

Sometimes it is difficult for `lpdoc` to distinguish include files and CIAO packages from normal *user* files (i.e., normal code files but which are not modules). The distinction is important because the former are quite different in their form of use (they are loaded via `include/1` or `use_package/1` declarations instead of `ensure_loaded/1`) and effect (since they are included, they 'export' operators, declarations, etc.), and should typically be documented differently. There is a special `comment/2` declaration which provides a way of defining the intended use of the file. This declaration is normally not needed in modules, include files, or packages, but should be added in user files (i.e., those meant to be loaded using `ensure_loaded/1`).

2.7.5 Splitting large documents into parts

As mentioned before, in `lpdoc` each documented file (each component) corresponds to a chapter in the generated manual. In large documents, it is sometimes convenient to build a super-structure of parts, each of which groups several chapters. There is a special value of the second argument of the `:- comment filetype, ...` declaration mentioned above designed for this purpose. The special *filetype* value `part` can be used to flag that the file in which it appears should be documented as the start of one of the major *parts in a large document*. In order to introduce such a part, a `.pl` file with a declaration `:- comment filetype, part` should be inserted in the sequence of files that make up the `COMPONENTS` variable of the `SETTINGS` file at each point in which a major part starts. The `:- comment title, "..."` declaration of this file will be used as the part title, and the `:- comment module, "..."` declaration text will be used as the introduction to the part.

2.7.6 Documenting reexported predicates

Reexported predicates, i.e., predicates which are exported by a module `m1` but defined in another module `m2` which is used by `m1`, are normally not documented in the original module,

but instead a simple reference is included to the module in which it is defined. This can be changed, so that the documentation is included in the original module, by using a `comment/2` declaration with `doinclude` in the first argument (see the `comments` library). This is often useful when documenting a library made of several components. For a simple user's manual, it is often sufficient to include in the `lpdoc SETTINGS` file the principal module, which is the one which users will do a `use_module/1` of, in the manual. This module typically exports or reexports all the predicates which define the library's user interface. Note, however, that currently, due to limitations in the implementation, only the comments inside assertions (but not those in `comment/2` declarations) are included for reexported predicates.

2.7.7 Separating the documentation from the source file

Sometimes one would not like to include long introductory comments in the module itself but would rather have them in a different file. This can be done quite simply by using the `@include` command. For example, the following declaration:

```
:- comment(module,"@include{Intro.lpd doc}").
```

will include the contents of the file `Description.lpd doc` as the module description.

Alternatively, sometimes one may want to generate the documentation from a completely different file. Assuming that the original module is `m1.pl`, this can be done by calling the module containing the documentation `m1_doc.pl`. This `m1_doc.pl` file is the one that will be included in the `lpdoc SETTINGS` file, instead of `m1.pl`. `lpdoc` recognizes and treats such `_doc` files specially so that the name without the `_doc` part is used in the different parts of the documentation, in the same way as if the documentation were placed in file `m1`.

2.7.8 Generating auxiliary files (e.g., READMEs)

Using `lpdoc` it is often possible to use a common source for documentation text which should appear in several places. For example, assume a file `INSTALL.lpd doc` contains text (with `lpdoc` formatting commands) describing an application. This text can be included in a section of the main file documentation as follows:

```
:- comment(module,"
...
@section{Installation instructions}
@include{INSTALL.lpd doc}
...
").
```

At the same time, this text can be used to generate a nicely formatted `INSTALL` file in `ascii`, which can perhaps be included in the top level of the source directory of the application. To this end, an `INSTALL.pl` file as follows can be constructed:

```
:- use_package([assertions]).
:- comment(title,"Installation instructions").
:- comment(module,"@include{INSTALL.lpd doc}").
main. %% forces file to be documented as an application
```

Then, the `ascii INSTALL` file can be generated by simply running `gmake ascii` in a directory with a `SETTINGS` file where `MAIN` is set to `INSTALL.pl`.

2.8 Troubleshooting

These are some common errors which may be found using `lpdoc` and the usual fix:

- Sometimes, messages of the type:

```
gmake: *** No rule to make target 'myfile.texic', needed by
'main.texic'.  Stop.
```

appear (i.e., in the case above when running `(g)make main.target`). Since `lpdoc` definitely knows how to make a `.texic` file given a `.pl` file, this means (in make's language) that it *cannot find the corresponding .pl file* (`myfile.pl` in the case above). The usual reason for this is that there is no directory path to this file declared in the `SETTINGS` file.

- Messages of the type:

```
! No room for a new @write .
```

while converting from `.texi` to `.dvi` (i.e., while running `tex`). These messages are `tex`'s way of saying that an internal area (typically for an index) is full. This is normally because more indices were selected in the `INDICES` variable of the `SETTINGS` file than the maximum number supported by the intalled version of `tex/ texinfo` installations, as mentioned in Section 2.1 [Generating a manual], page 11. The easiest fix is to reduce the number of indices generated. Alternatively, it may be possible to recompile your local `tex/ texinfo` installation with a higher number of indices.

- Missing links in `info` files (a section which exists in the printed document cannot be accessed in the on-line document) can be due to the presence of a colon (:), a comma (,), a double dash (--), or other such separators in a section name. Due to limitations of `info` section names cannot contain these symbols.
- Menu listings in `info` which *do not work* (i.e., the menu listings are there, but they cannot be followed): see if they are indented. In that case it is due to an `itemize` or `enumerate` which was not closed.

3 Enhancing Documentation with Machine-Readable Comments

Author(s): Manuel Hermenegildo.

Version: 1.9#29 (1999/11/24, 16:47:1 MET)

Version of last change: 1.9#25 (1999/11/23, 0:39:18 MET)

This defines the admissible uses of the `comment/2` declaration (which is used mainly for adding machine readable comments to programs), the formatting commands which can be used in the text strings inside these comments, and some related properties and data types. These declarations are ignored by the compiler in the same way as classical comments. Thus, they can be used to document the program source in place of (or in combination with) the normal comments typically inserted in the code by programmers. However, because they are more structured and they are machine-readable, they can also be used to generate printed or on-line documentation automatically, using the `lpdoc` automatic documentation generator. These *textual comments* are meant to be complementary to the formal statements present in *assertions* (see the `assertions` library).

3.1 Usage and interface (comments)

- **Library usage:**

It is not necessary to use this library in user programs. The recommended procedure in order to make use of the `comment/2` declarations that this library defines is to include instead the `assertions` package, which provides efficient support for all assertion- and comment-related declarations, using one of the following declarations, as appropriate:

```
:- module(...,[assertions]).
:- use_package(assertions).
```

- **Exports:**

- *Properties:*
`docstring/1, stringcommand/1.`
- *Regular Types:*
`version_descriptor/1, filetype/1.`

- **Other modules used:**

- *System library modules:*
`dcg_expansion.`
- *Internal (engine) modules:*
`arithmetic, atomic_basic, attributes, basic_props, basiccontrol, concurrency, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_basic, term_compare, term_typing.`

3.2 Documentation on exports (comments)

`comment/2:`

DECLARATION

This declaration provides one of the main means for adding *machine readable comments* to programs (the other one is adding *documentation strings* to assertions).

docstring/1:

PROPERTY

Defines the format of the character strings which can be used in machine readable comments (`comment/2` declarations) and assertions. These character strings can include certain *formatting commands*.

- All printable characters are admissible in documentation strings except “@”, “{,” and “}”. To produce these characters the following *escape sequences* should be used, respectively: `@@`, `@{`, and `@}`.
- In order to allow better formatting of on-line and printed manuals, in addition to normal text, certain formatting commands can be used within these strings. The syntax of all these commands is:

`@command`

(followed by either a space or `{}`), or

`@command{body}`

where *command* is the command name and *body* is the (possibly empty) command body.

The set of commands currently admitted can be found in the documentation for the predicate `stringcommand/1`.

Usage: `docstring(Text)`

- *Description:* `Text` is a *documentation string*.

stringcommand/1:

PROPERTY

Defines the set of structures which can result from parsing a formatting command admissible in comment strings inside assertions.

In order to make it possible to produce documentation in a wide variety of formats, the command set is kept small. The names of the commands are intended to be reminiscent of the commands used in the LaTeX text formatting system, except that “@” is used instead of “\.” Note that \ would need to be escaped in ISO-Prolog strings, which would make the source less readable (and, in any case, many ideas in LaTeX were taken from scribe, where the escape character was indeed @!).

The following are the currently admissible commands.

- **Indexing commands:**

The following commands are used to mark certain words or sentences in the text as concepts, names of predicates, libraries, files, etc. The use of these commands is highly recommended, since it results in very useful indices with little effort.

`@index{text}`

text will be printed in an emphasized font and will be included in the concept definition index (and also in the usage index). This command should be used for the first or *definitional* appearance(s) of a concept. The idea is that the concept definition index can be used to find the definition(s) of a concept.

`@cindex{text}`

text will be included in the concept index (and also in the usage index), but it is not printed. This is used in the same way as above, but allows sending to the index a different text than the one that is printed in the text.

`@concept{text}`

text will be printed (in a normal font). This command is used to mark that some text is a defined concept. In on-line manuals, a direct access

to the corresponding concept definition may also be generated. A pointer to the place in which the `@concept` command occurs will appear only in the usage index.

@pred{*predname*}

predname (which should be in functor/arity form) is the name of a predicate and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a predicate (or a property or type) in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding predicate definition may also be generated.

@op{*operatorname*}

operatorname (which should be in functor/arity form) is the name of an operator and will be printed in fixed-width, typewriter-like font. This command should be used when referring to an operator in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding operator definition may also be generated.

@decl{*declname*}

declname (which should be in functor/arity form) is the name of a declaration and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a declaration in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding declaration definition may also be generated.

@lib{*libname*}

libname is the name of a library and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a module or library in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding module definition may also be generated.

@apl{*aplname*}

aplname is the name of an application and will be printed in fixed-width, typewriter-like font. This command should be used when referring to an application in a documentation string. A reference will be included in the usage index.

@file{*filename*}

filename is the name of a file and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a file in a documentation string. A reference will be included in the usage index.

@var{*varname*}

varname is the name of a variable and will be formatted in an emphasized font. Note that when referring to variable names in a `pred/1` declaration, such names should be enclosed in `@var` commands for the automatic documentation system to work correctly.

- **Referencing commands:**

The following commands are used to introduce *bibliographic citations* and *references* to *sections*, *urls*, *email addresses*, etc.

@cite{*keyword*}

keyword is the identifier of a *bibliographic entry*. Such entry is assumed to reside in one of a number of `bibtex` files (*.bib files*). A reference in

brackets ([]) is inserted in the text and the full reference is included at the end, with all other references, in an appendix. For example, `@cite{iso-prolog}` will introduce a citation to a bibliographic entry whose keyword is `iso-prolog`. The list of bibliography files which will be searched for a match is determined by the `BIBFILES` variable of the `lpdoc SETTINGS` file.

`@ref{section title}`

introduces at point a reference to the section or node *section title*, where *section title* must be the exact *text* of the section title.

`@uref{URL}`

introduces at point a reference to the *Universal Resource Locator* (i.e., a *WWW address* 'URL'.

`@uref{text}{URL}`

introduces at point a reference to the Universal Resource Locator URL, associated to the text *text*.

`@email{address}`

introduces at point a reference to *email address* *address*.

`@email{text}{address}`

introduces at point a reference to the email address *address*, associated to the text *text*.

- **Formatting commands:**

The following commands are used to format certain words or sentences in a special font, build itemized lists, introduce sections, include examples, etc.

`@comment{text}`

text will be treated as a *comment* and will be ignored.

`@begin{itemize}`

marks the beginning of an *itemized list*. Each item should be in a separate paragraph and preceded by an `@item` command.

`@item`

marks the beginning of a new *item in an itemized list*.

`@end{itemize}`

marks the end of an itemized list.

`@begin{enumerate}`

marks the beginning of an *enumerated list*. Each item should be in a separate paragraph and preceded by an `@item` command.

`@end{enumerate}`

marks the end of an enumerated list.

`@begin{description}`

marks the beginning of a *description list*, i.e., a list of items and their description (this list describing the different allowable commands is in fact a description list). Each item should be in a separate paragraph and contained in an `@item{itemtext}` command.

`@item{itemtext}`

marks the beginning of a *new item in description list*, and contains the header for the item.

`@end{description}`


marks the end of a description list.

<code>@begin{verbatim}</code>	marks the beginning of <i>fixed format text</i> , such as a program example. A fixed-width, typewriter-like font is used.
<code>@end{verbatim}</code>	marks the end of formatted text.
<code>@begin{cartouche}</code>	marks the beginning of a section of text in a <i>framed box</i> , with round corners.
<code>@end{cartouche}</code>	marks the end of a section of text in a framed box.
<code>@section{text}</code>	starts a <i>section</i> whose title is <i>text</i> . Due to a limitation of the <code>info</code> format, do not use <code>:</code> or <code>-</code> or <code>,</code> in section, subsection, title (chapter), etc. headings.
<code>@subsection{text}</code>	starts a <i>subsection</i> whose title is <i>text</i> .
<code>@footnote{text}</code>	places <i>text</i> in a <i>footnote</i> .
<code>@today</code>	prints the current <i>date</i> .
<code>@hfill</code>	introduces horizontal filling space (may be ignored in certain formats).
<code>@bf{text}</code>	<i>text</i> will be formatted in <i>bold face</i> or any other <i>strong face</i> .
<code>@em{text}</code>	<i>text</i> will be formatted in <i>italics face</i> or any other <i>emphasis face</i> .
<code>@tt{text}</code>	<i>text</i> will be formatted in a <i>fixed-width font</i> (i.e., <i>typewriter-like font</i>).
<code>@key{key}</code>	<i>key</i> is the identifier of a <i>keyboard key</i> (i.e., a letter such as <code>a</code> , or a special key identifier such as <code>RET</code> or <code>DEL</code>) and will be formatted as <code><u>LFD</u></code> or in a fixed-width, typewriter-like font.
<code>@sp{N}</code>	generates <i>N blank lines</i> of space. Forces also a paragraph break.
<code>@p</code>	forces a <i>paragraph break</i> , in the same way as leaving one or more blank lines.
<code>@noindent</code>	used at the beginning of a paragraph, states that the first line of the paragraph should not be indented. Useful, for example, for <i>avoiding indentation</i> on paragraphs that are continuations of other paragraphs, such as after a <code>verbatim</code> .

- **Accents and special characters:**

The following commands can be used to insert *accents* and *special characters*.

<code>@' {o}</code>	\Rightarrow ò
<code>@' {o}</code>	\Rightarrow ó
<code>@~ {o}</code>	\Rightarrow ô
<code>@. . {o}</code>	\Rightarrow ö
<code>@~ {o}</code>	\Rightarrow õ
<code>@= {o}</code>	\Rightarrow ô
<code>@. {o}</code>	\Rightarrow ò

@u{o}	⇒ Ů
@v{o}	⇒ ǝ
@H{o}	⇒ Ǟ
@t{oo}	⇒ ǝǝ
@c{o}	⇒ ȳ
@d{o}	⇒ ȳ
@b{o}	⇒ ȳ
@oe	⇒ œ
@OE	⇒ Œ
@ae	⇒ æ
@AE	⇒ Æ
@aa	⇒ ȁ
@AA	⇒ Å
@o	⇒ ø
@O	⇒ Ø
@l	⇒ ł
@L	⇒ Ł
@ss	⇒ ß
@?	⇒ ¿
@!	⇒ ¡
@i	⇒ í
@j	⇒ ÿ
@copyright	⇒ ©
@iso	⇒ 
@bullet	⇒ •
@result	⇒ ⇒

- **Inclusion commands:**

The following commands are used to include code or strings of text as part of documentation. The latter may reside in external files or in the file being documented. The former must be part of the module being documented. There are also commands for inserting and scaling images.

@include{*filename*}

the contents of *filename* will be included in-line, as if they were part of the string. This is useful for common pieces of documentation or storing in a separate file long explanations if they are perceived to clutter the source file.

@includeverbatim{*filename*}

as above, but the contents of the file are included verbatim, i.e., commands within the file are not interpreted. This is useful for including code examples which may contain @'s, etc.

@includefact{*factname*}

it is assumed that the file being documented contains a fact of the predicate *factname*/1, whose argument is a character string. The contents of that character string will be included in-line, as if they were part of the documentation string. This is useful for *sharing pieces of text* between the documentation and the running code. An example is the text which explains the *usage of a command* (options, etc.).

@includedef{*predname*}

it is assumed that the file being documented contains a definition for the predicate *predname*. The clauses defining this predicate will be included in-line, in verbatim mode, as if they were part of the documentation string.

@image{*epsfile*}

including an image at point, contained in file *epsfile*. The *image file* should be in *encapsulated postscript* format.

@image{*epsfile*}{*width*}{*height*} same as above, but *width* and *height* should be integers which provide a size (in points) to which the image will be scaled.

Usage: stringcommand(C0)

- *Description:* C0 is a structure denoting a command that is admissible in strings inside assertions.

version_descriptor/1:

REGTYPE

A structure denoting a complete version description:

```
version_descriptor(version(Version,Date)) :-
    version_number(Version),
    ymd_date(Date).
version_descriptor(version(Version,Date,Time)) :-
    version_number(Version),
    ymd_date(Date),
    time_struct(Time).
```

Usage: version_descriptor(Descriptor)

- *Description:* Descriptor is a complete version descriptor.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

filetype/1:

REGTYPE

Intended uses of a file:

```
filetype(module).
filetype(user).
filetype(include).
filetype(package).
filetype(part).
```

Usage: filetype(Type)

- *Description:* Type describes the intended use of a file.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

3.3 Documentation on internals (comments)

version_number/1: REGTYPE

Version is a structure denoting a complete version number (major version, minor version, and patch number):

```
version_number(Major*Minor+Patch) :-
    int(Major),
    int(Minor),
    int(Patch).
```

Usage: version_number(Version)

- *Description:* Version is a complete version number
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

ymd_date/1: REGTYPE

A Year/Month/Day structure denoting a date:

```
ymd_date(Y/M/D) :-
    int(Y),
    int(M),
    int(D).
```

Usage: ymd_date(Date)

- *Description:* Date is a Year/Month/Day structure denoting a date.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

time_struct/1: REGTYPE

A struture containing time information:

```
time_struct(Hours:Minutes*Seconds+TimeZone) :-
    int(Hours),
    int(Minutes),
    int(Seconds),
    atm(TimeZone).
```

Usage: time_struct(Time)

- *Description:* Time contains time information.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

version_maintenance_type/1: REGTYPE

Possible kinds of version maintenance for a file:

```
version_maintenance_type(on).
version_maintenance_type(off).
version_maintenance_type(dir(Path)) :-
    atm(Path).
```

- **on**: version numbering is maintained locally in the file in which the declaration occurs, i.e., an independent version number is kept for this file and the current version is given by the most recent `comment(version(...),...)` declaration.
- **off**: no version numbering maintained.
- **dir(Path)**: version numbering is maintained (globally) in directory **Path**. This is useful for maintaining a common global version for an application which involves several files.

The automatic maintenance of version numbers is typically done by the CIAO `emacs` mode.

Usage: `version_maintenance_type(Type)`

- *Description:* **Type** a type of version maintenance for a file.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

4 Program assertions

Author(s): The CLIP Group, clip@dia.fi.upm.es, <http://www.clip.dia.fi.upm.es/>, Facultad de Informática, Universidad Politécnica de Madrid.

Version: 1.3#124 (1999/11/27, 4:4:5 MET)

The **assertions** package adds a number of new declaration definitions and new operator definitions which allow including program assertions in user programs. Such assertions can be used to describe predicates, properties, modules, applications, etc. These descriptions can be formal specifications (such as preconditions and post-conditions) or machine-readable textual comments.

This module is part of the **assertions** library. It defines the basic code-related assertions, i.e., those intended to be used mainly by compilation-related tools, such as the static analyzer or the run-time test generator.

Giving specifications for predicates and other program elements is the main functionality documented here. The exact syntax of comments is described in the autodocumenter (`lpdoc` [Knu84,Her99]) manual, although some support for adding machine-readable comments in assertions is also mentioned here.

There are two kinds of assertions: predicate assertions and program point assertions. All predicate assertions are currently placed as directives in the source code, i.e., preceded by “:-”. Program point assertions are placed as goals in clause bodies.

4.1 More info

The facilities provided by the library are documented in the description of its component modules. This documentation is intended to provide information only at a “reference manual” level. For a more tutorial introduction to the subject and some more examples please see the document “An Assertion Language for Debugging of Constraint Logic Programs (Technical Report CLIP2/97.1)”. The assertion language implemented in this library is modeled after this design document, although, due to implementation issues, it may differ in some details. The purpose of this manual is to document precisely what the implementation of the library supports at any given point in time.

4.2 Some attention points

- **Formatting commands within text strings:** many of the predicates defined in these modules include arguments intended for providing textual information. This includes titles, descriptions, comments, etc. The type of this argument is a character string. In order for the automatic generation of documentation to work correctly, this character string should adhere to certain conventions. See the description of the `docstring/1` type/grammar for details.
- **Referring to variables:** In order for the automatic documentation system to work correctly, variable names (for example, when referring to arguments in the head patterns of *pred* declarations) must be surrounded by an `@var` command. For example, `@var{VariableName}` should be used for referring to the variable “VariableName”, which will appear then formatted as follows: `VariableName`. See the description of the `docstring/1` type/grammar for details.

4.3 Usage and interface (assrt)

- **Library usage:**

The recommended procedure in order to make use of assertions in user programs is to include the `assertions` syntax library, using one of the following declarations, as appropriate:

```
:- module(...,[assertions]).
:- include(library(assertions)).
:- use_package([assertions]).
```

- **Exports:**

- *Predicates:*
`check/1, trust/1, true/1, false/1.`

- **New operators defined:**

```
=>/2 [975,xfx], ::/2 [978,xfx], decl/1 [1150,fx], decl/2 [1150,xfx], pred/1 [1150,fx], pred/2
[1150,xfx], prop/1 [1150,fx], prop/2 [1150,xfx], modedef/1 [1150,fx], calls/1 [1150,fx],
calls/2 [1150,xfx], success/1 [1150,fx], success/2 [1150,xfx], comp/1 [1150,fx], comp/2
[1150,xfx], entry/1 [1150,fx], regtype/1 [1150,fx], regtype/2 [1150,xfx].
```

- **New declarations defined:**

```
pred/1, pred/2, calls/1, calls/2, success/1, success/2, comp/1, comp/2, prop/1,
prop/2, regtype/1, type/1, regtype/2, type/2, entry/1, modedef/1, decl/1, decl/2,
comment/2.
```

- **Other modules used:**

- *System library modules:*
`assertions/assrt_props.`
- *Internal (engine) modules:*
`arithmetic, atomic_basic, attributes, basic_props, basiccontrol, concurrency,
data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic,
system_info, term_basic, term_compare, term_typing.`

4.4 Documentation on new declarations (assrt)

pred/1:

DECLARATION

Usage: `:- pred(AssertionBody).`

- *Description:* This assertion provides information on a predicate. The body of the assertion (`AssertionBody`) contains properties or comments in the formats defined by `assrt_body/1`.

More than one of these assertions may appear per predicate, in which case each one represents a possible “mode” of use (usage) of the predicate. The exact scope of the usage is defined by the properties given for calls in the body of each assertion (which should thus distinguish the different usages intended).

- *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (`assrt_body/1`)

pred/2:

DECLARATION

Usage: `:- pred(AssertionStatus,AssertionBody).`

- *Description:* This assertion is similar to a **pred/1** assertion but it is explicitly qualified. Non-qualified **pred/1** assertions are assumed the qualifier **check**.
- *The following properties should hold at call time:*
AssertionStatus is an acceptable status for an assertion. (assrt_status/1)
AssertionBody is an assertion body. (assrt_body/1)

calls/1: DECLARATION

Usage: :- calls(**AssertionBody**).

- *Description:* This assertion is similar to a **pred/1** assertion but it only provides information about the calls to a predicate.
- *The following properties should hold at call time:*
AssertionBody is a call assertion body. (c_assrt_body/1)

calls/2: DECLARATION

Usage: :- calls(**AssertionStatus**,**AssertionBody**).

- *Description:* This assertion is similar to a **calls/1** assertion but it is explicitly qualified. Non-qualified **calls/1** assertions are assumed the qualifier **check**.
- *The following properties should hold at call time:*
AssertionStatus is an acceptable status for an assertion. (assrt_status/1)
AssertionBody is a call assertion body. (c_assrt_body/1)

success/1: DECLARATION

Usage: :- success(**AssertionBody**).

- *Description:* This assertion is similar to a **pred/1** assertion but it only provides information about the answers to a predicate.
- *The following properties should hold at call time:*
AssertionBody is a predicate assertion body. (s_assrt_body/1)

success/2: DECLARATION

Usage: :- success(**AssertionStatus**,**AssertionBody**).

- *Description:* This assertion is similar to a **success/1** assertion but it is explicitly qualified. Non-qualified **success/1** assertions are assumed the qualifier **check**.
- *The following properties should hold at call time:*
AssertionStatus is an acceptable status for an assertion. (assrt_status/1)
AssertionBody is a predicate assertion body. (s_assrt_body/1)

comp/1: DECLARATION

Usage: :- comp(**AssertionBody**).

- *Description:* This assertion is similar to a **pred/1** assertion but it only provides information about the global execution properties of a predicate.
- *The following properties should hold at call time:*
AssertionBody is a comp assertion body. (g_assrt_body/1)

comp/2:

DECLARATION

Usage: `:- comp(AssertionStatus,AssertionBody).`

- *Description:* This assertion is similar to a `comp/1` assertion but it is explicitly qualified. Non-qualified `comp/1` assertions are assumed the qualifier `check`.
- *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (assrt_status/1)

`AssertionBody` is a `comp` assertion body. (g_assrt_body/1)

prop/1:

DECLARATION

Usage: `:- prop(AssertionBody).`

- *Description:* This assertion is similar to a `pred` assertion but it flags that the predicate being documented is also a “property.” Properties are standard predicates, but which are *deterministic* and *guaranteed to terminate for any possible instantiation state of their argument(s)* (and can thus be safely used as run-time checks).

The set of properties is thus a strict subset of the set of predicates. Note that properties can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

- *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (assrt_body/1)

prop/2:

DECLARATION

Usage: `:- prop(AssertionStatus,AssertionBody).`

- *Description:* This assertion is similar to a `prop/1` assertion but it is explicitly qualified. Non-qualified `prop/1` assertions are assumed the qualifier `check`.
- *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (assrt_status/1)

`AssertionBody` is an assertion body. (assrt_body/1)

regtype/1:

DECLARATION

Usage: `:- regtype(AssertionBody).`

- *Description:* This assertion is similar to a `pred` assertion but it flags that the predicate being documented is also a “regular type.” This allows for example checking whether it is in the class of types supported by the type checking and inference modules. Currently, types are properties whose definitions are *regular programs*.

A regular program is defined by a set of clauses, each of the form:

$$p(x, v_1, \dots, v_n) \text{ :- } body_1, \dots, body_k.$$

where:

1. x is a term whose variables (which are called *term variables*) are unique, i.e., it is not allowed to introduce equality constraints between the variables of x .
For example, `p(f(X, Y)) :- ...` is valid, but `p(f(X, X)) :- ...` is not.
2. in all clauses defining `p/n+1` the terms x do not unify except maybe for one single clause in which x is a variable.
3. $n \geq 0$ and `p/n` is a *parametric type functor* (whereas the predicate defined by the clauses is `p/n+1`).

4. v_1, \dots, v_n are unique variables, which are called *parametric variables*.
5. Each `body_i` is of the form:
 1. `regtype(z, t)` where z is one of the *term variables* and t is a *regular type expression*;
 2. `q(y, t_1, \dots, t_m)` where $m \geq 0$, q/m is a *parametric type functor*, not in the set of functors `=/2, ^/2, ./3 regtype/2`.
 t_1, \dots, t_m are *regular type expressions*, and y is a *term variable*.
6. Each term variable occurs at most once in the clause's body (and should be as the first argument of a literal).

A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables.

A parametric type functor is a regular type, defined by a regular program, or a basic type. Basic types are defined in the CIAO `engine(basic_props)` manual.

The set of types is thus a well defined subset of the set of properties. Note that types can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

- *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (assrt_body/1)

type/1: DECLARATION

Usage:

- *Description:* Same as `regtype/1`. Deprecated. Included only for backwards compatibility.

regtype/2: DECLARATION

Usage: `:- regtype(AssertionStatus,AssertionBody).`

- *Description:* This assertion is similar to a `regtype/1` assertion but it is explicitly qualified. Non-qualified `regtype/1` assertions are assumed the qualifier `check`.
- *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (assrt_status/1)

`AssertionBody` is an assertion body. (assrt_body/1)

type/2: DECLARATION

Usage:

- *Description:* Same as `regtype/2`. Deprecated. Included only for backwards compatibility.

entry/1: DECLARATION

Usage: `:- entry(AssertionBody).`

- *Description:* This assertion provides information about the calls to a predicate. It is identical syntactically to a `calls/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. As a

result, if these assertions are erroneous they can introduce bugs in programs. Thus, `entry/1` assertions should be written with care.

An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program. The main use is in providing information on the ways in which exported predicates of a module will be called from outside the module. This will greatly improve the precision of the analyzer, which otherwise has to assume that the arguments that exported predicates receive are any arbitrary term.

- *The following properties should hold at call time:*

`AssertionBody` is a call assertion body. (`c_assrt_body/1`)

`modedef/1:`

DECLARATION

Usage: `:- modedef(AssertionBody).`

- *Description:* This assertion is used to define modes. A mode defines in a compact way a set of call and success properties. Once defined, modes can be applied to predicate arguments in assertions. The meaning of this application is that the call and success properties defined by the mode hold for the argument to which the mode is applied. Thus, a mode is conceptually a “property macro”.

The syntax of mode definitions is similar to that of `pred` declarations.

Example:

The following set of assertions:

```
:- modedef +A : nonvar(A) # "A is bound upon predicate entry."
```

```
:- pred p(+A,B) : integer(A) => ground(B).
```

is equivalent to:

```
:- pred p(A,B) : (nonvar(A),integer(A)) => ground(B)
   # "A is bound upon predicate entry."
```

- *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (`assrt_body/1`)

`decl/1:`

DECLARATION

Usage: `:- decl(AssertionBody).`

- *Description:* This assertion is similar to a `pred/1` assertion but it is used for declarations instead than for predicates.
- *The following properties should hold at call time:*

`AssertionBody` is an assertion body. (`assrt_body/1`)

`decl/2:`

DECLARATION

Usage: `:- decl(AssertionStatus,AssertionBody).`

- *Description:* This assertion is similar to a `decl/1` assertion but it is explicitly qualified. Non-qualified `decl/1` assertions are assumed the qualifier `check`.
- *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assrt_status/1`)

`AssertionBody` is an assertion body. (`assrt_body/1`)

comment/2:

DECLARATION

Usage: `comment(Pred, Comment).`

- *Description:* This assertion gives a text `Comment` for a given predicate `Pred`.
- *The following properties should hold at call time:*

`Pred` is a head pattern.

(head_pattern/1)

`Comment` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments. (`docstring/1`)

4.5 Documentation on exports (assrt)

check/1:

PREDICATE

Usage: `check(PropertyConjunction)`

- *Description:* This assertion provides information on a clause program point (position in the body of a clause). Calls to a `check/1` assertion can appear in the body of a clause in any place where a literal can normally appear. The property defined by `PropertyConjunction` should hold in all the run-time stores corresponding to that program point.
- *The following properties should hold at call time:*

`PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`property_conjunction/1`)

trust/1:

PREDICATE

Usage: `trust(PropertyConjunction)`

- *Description:* This assertion also provides information on a clause program point. It is identical syntactically to a `check/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. While the compiler may in some cases detect an inconsistency between a `trust/1` assertion and the program, in all other cases the information given in the assertion will be taken to be true. As a result, if these assertions are erroneous they can introduce bugs in programs. Thus, `trust/1` assertions should be written with care.

An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program (either because the information is not present or because the analyzer being used is not precise enough). In particular, providing information on external predicates which may not be accessible at the time of compiling the module can greatly improve the precision of the analyzer. This can be easily done with trust assertion.

- *The following properties should hold at call time:*

`PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`property_conjunction/1`)

true/1:

PREDICATE

Usage: true(PropertyConjunction)

- *Description:* This assertion is identical syntactically to a **check/1** assertion. However, the properties stated have been proved to hold by the analyzer. Thus, these assertions often represent the analyzer output.
- *The following properties should hold at call time:*

PropertyConjunction is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (property_conjunction/1)

false/1:

PREDICATE

Usage: false(PropertyConjunction)

- *Description:* This assertion is identical syntactically to a **check/1** assertion. However, the properties stated have been proved not to hold by the analyzer. Thus, these assertions often represent the analyzer output.
- *The following properties should hold at call time:*

PropertyConjunction is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (property_conjunction/1)

5 Types and properties related to assertions

Author(s): Manuel Hermenegildo.

Version: 1.3#124 (1999/11/27, 4:4:5 MET)

Version of last change: 0.9#98 (1999/5/25, 17:1:28 MEST)

This module is part of the `assertions` library. It defines types and properties related to assertions.

5.1 Usage and interface (`assrt_props`)

- **Library usage:**
`:- use_module(library(assrt_props)).`
- **Exports:**
 - *Properties:*
`head_pattern/1, docstring/1.`
 - *Regular Types:*
`assrt_body/1, complex_arg_property/1, property_conjunction/1, property_starterterm/1, complex_goal_property/1, c_assrt_body/1, s_assrt_body/1, g_assrt_body/1, assrt_status/1, assrt_type/1, predfunctor/1, propfunctor/1.`
- **Other modules used:**
 - *System library modules:*
`dcg_expansion.`
 - *Internal (engine) modules:*
`arithmetic, atomic_basic, attributes, basic_props, basiccontrol, concurrency, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_basic, term_compare, term_typing.`

5.2 Documentation on exports (`assrt_props`)

assrt_body/1:

REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `func/1`, etc. assertions. Such a body is of the form:

`Pr [:: DP] [: CP] [=> AP] [+ GP] [# CO]`

where (fields between [...] are optional):

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `DP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which expresses properties which are compatible with the predicate, i.e., instantiations made by the predicate are *compatible* with the properties in the sense that applying the property at any point to would not make it fail.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.

- AP is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
- GP is a (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- CO is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`). See the lpdoc manual for documentation on assertion comments.

Usage: `assrt_body(X)`

- *Description:* X is an assertion body.
- *The following properties should hold globally:*
Defines a regular type.

(`regtype/1`)

head_pattern/1:

PROPERTY

A head pattern can be a predicate name (functor/arity) (`predname/1`) or a term. Thus, both `p/3` and `p(A,B,C)` are valid head patterns. In the case in which the head pattern is a term, each argument of such a term can be:

- A variable. This is useful in order to be able to refer to the corresponding argument positions by name within properties and in comments. Thus, `p(Input,Parameter,Output)` is a valid head pattern.
- A ground term. In this case this term determines a property of the corresponding argument. The actual property referred to is that given by the term but with one more argument added at the beginning, which is a new variable which, in a rewriting of the head pattern, appears at the argument position occupied by the term. Unless otherwise stated (see below), the property built this way is understood to hold for both calls and answers. For example, the head pattern `p(Input,list(integer),Output)` is valid and equivalent for example to having the head pattern `p(Input,A,Output)` and stating that the property `list(A,integer)` holds for the calls and successes of the predicate.
- Finally, it can also be a variable or a ground term, as above, but preceded by a “mode.” This mode determines in a compact way certain call or answer properties. For example, the head pattern `p(Input,+list(integer),Output)` is valid, as long as `+/1` is declared as a mode.

Acceptable modes are documented in `library(modes)`. User defined modes are documented in `modedef/1`.

Usage: `head_pattern(Pr)`

- *Description:* Pr is a head pattern.

complex_arg_property/1:

REGTYPE

`complex_arg_property(Props)`

Props is a (possibly empty) complex argument property. Such properties can appear in two formats, which are defined by `property_conjunction/1` and `property_starterm/1` respectively. The two formats can be mixed provided they are not in the same field of an assertion. I.e., the following is a valid assertion:

```
:- pred foo(X,Y) : nonvar * var => (ground(X),ground(Y)).
```

Usage: `complex_arg_property(Props)`

- *Description:* **Props** is a (possibly empty) complex argument property
- *The following properties should hold globally:*
Defines a regular type. (regtype/1)

property_conjunction/1: REGTYPE

This type defines the first, unabridged format in which properties can be expressed in the bodies of assertions. It is essentially a conjunction of properties which refer to variables. The following is an example of a complex property in this format:

- `(integer(X),list(Y,integer))`: X has the property `integer/1` and Y has the property `list/2`, with second argument `integer`.

Usage: `property_conjunction(Props)`

- *Description:* **Props** is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument.
- *The following properties should hold globally:*
Defines a regular type. (regtype/1)

property_starterm/1: REGTYPE

This type defines a second, compact format in which properties can be expressed in the bodies of assertions. A `property_starterm/1` is a term whose main functor is `*/2` and, when it appears in an assertion, the number of terms joined by `*/2` is exactly the arity of the predicate it refers to. A similar series of properties as in `property_conjunction/1` appears, but the arity of each property is one less: the argument position to which they refer (first argument) is left out and determined by the position of the property in the `property_starterm/1`. The idea is that each element of the `*/2` term corresponds to a head argument position. Several properties can be assigned to each argument position by grouping them in curly brackets. The following is an example of a complex property in this format:

- `integer * list(integer)`: the first argument of the procedure (or function, or ...) has the property `integer/1` and the second one has the property `list/2`, with second argument `integer`.
- `{integer,var} * list(integer)`: the first argument of the procedure (or function, or ...) has the properties `integer/1` and `var/1` and the second one has the property `list/2`, with second argument `integer`.

Usage: `property_starterm(Props)`

- *Description:* **Props** is either a term or several terms separated by `*/2`. The main functor of each of those terms corresponds to that of the definition of a property, and the arity should be one less than in the definition of such property. All arguments of each such term are ground.
- *The following properties should hold globally:*
Defines a regular type. (regtype/1)

complex_goal_property/1: REGTYPE

`complex_goal_property(Props)`

Props is a (possibly empty) complex goal property. Such properties can be either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. Such properties apply to all executions of all goals of the predicate which comply with the assertion in which the **Props** appear.

The arguments of the terms in **Props** are implicitly augmented with a first argument which corresponds to a goal of the predicate of the assertion in which the **Props** appear. For example, the assertion

```
:- comp var(A) + not_further_inst(A).
```

has property `not_further_inst/1` as goal property, and establishes that in all executions of `var(A)` it should hold that `not_further_inst(var(A),A)`.

Usage: `complex_goal_property(Props)`

- *Description:* **Props** is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. A first implicit argument in such terms identifies goals to which the properties apply.
- *The following properties should hold globally:*
Defines a regular type. (regtype/1)

c_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `call/1`, `entry/1`, etc. assertions. The following are admissible:

```
Pr : CP [# CO]
```

where (fields between [...] are optional):

- **CP** is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- **CO** is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `c_assrt_body(X)`

- *Description:* **X** is a call assertion body.
- *The following properties should hold globally:*
Defines a regular type. (regtype/1)

s_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `func/1`, etc. assertions. The following are admissible:

```
Pr : CP => AP # CO
Pr : CP => AP
Pr => AP # CO
Pr => AP
```

where:

- **Pr** is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- **CP** is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- **AP** is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
- **CO** is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `s_assrt_body(X)`

- *Description:* **X** is a predicate assertion body.
- *The following properties should hold globally:*

Defines a regular type.

(`regtype/1`)

g_assrt_body/1:

REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `comp/1` assertions. The following are admissible:

```
Pr : CP + GP # CO
Pr : CP + GP
Pr + GP # CO
Pr + GP
```

where:

- **Pr** is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- **CP** is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- **GP** contains (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- **CO** is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `g_assrt_body(X)`

- *Description:* **X** is a comp assertion body.
- *The following properties should hold globally:*

Defines a regular type.

(`regtype/1`)

assrt_status/1:

REGTYPE

The types of assertion status. They have the same meaning as the program-point assertions, and are as follows:


```

assrt_status(true).
assrt_status(false).
assrt_status(check).
assrt_status(checked).
assrt_status(trust).

```

Usage: `assrt_status(X)`

- *Description:* X is an acceptable status for an assertion.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

assrt_type/1:

REGTYPE

The admissible kinds of assertions:

```

assrt_type(pred).
assrt_type(prop).
assrt_type(decl).
assrt_type(func).
assrt_type(calls).
assrt_type(success).
assrt_type(comp).
assrt_type(entry).
assrt_type(modedef).

```

Usage: `assrt_type(X)`

- *Description:* X is an admissible kind of assertion.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

predfunctor/1:

REGTYPE

Usage: `predfunctor(X)`

- *Description:* X is a type of assertion which defines a predicate.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

propfunctor/1:

REGTYPE

Usage: `propfunctor(X)`

- *Description:* X is a type of assertion which defines a *property*.
- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

docstring/1:

PROPERTY

Usage: `docstring(String)`

- *Description:* `String` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments.

6 An Example - Documenting a Library Module

A simple example of the use of `lpdoc` is this manual, which can be built in the `doc` directory of the `lpdoc` distribution. Other examples of manuals generated using `lpdoc` can be found in the `CIAO` system and preprocessor `doc` directories (i.e., most of the `CIAO` manuals are generated using `lpdoc`). Some simpler examples can be found in the `examples` directory of the `lpdoc` distribution. In particular, the chapter following this one contains the documentation generated automatically for the module defined by file `examples/example_module.pl` (which for simplicity contains only assertions, i.e., no actual code) and which is included in source form below. Comparing this code with the output in the following chapter illustrates the use and some of the capabilities of `lpdoc`:

```
%% The module headers produce documentation on the module interface
%% Exported predicates (+ properties and types) are documented by default
:- module(example_module,
    [bar/1,baz/1,aorb/1,tree_of/2,list_or_aorb/2,list/1,q/1,q/2,r/1,
     og/1,t/5,u/3,w/1,p/5,long/1],
    [assertions,basicmodes,functions,regtypes]).

%% We import two types: list/1 and list/2 (now in basic_props, which is
%% exported by default from assertions).

%% We reexport list/1
:- reexport(library('engine/basic_props'),[ list/1 ]).

:- use_module('bar').
:- ensure_loaded(foo).

%% "comment" declarations provide additional information
:- comment(title,"Auto Documenter Output for the Example Module").

:- comment(author,"Alan Robinson").
:- comment(author,"David H.D. Warren").

:- comment(summary,"This is a brief summary description of the module
    or file. In this case the file is a library.").

:- comment(module,"This is where general comments on the file go. In
    this case the file is a library which contains some assertion examples
    for testing the @em{automatic documentation system}. ").

%% An example of a comment documenting a bug
:- comment(bug,"Library is hard to execute: no actual code!").

%% Standard declarations are documented with the corresponding predicate
:- data r/1.
:- dynamic q/2.
:- multifile p/3.
:- dynamic p/3.
:- meta_predicate p(?,.,?).

%% Uncommenting this would make these not appear in the documentation
%% :- comment(hide,[bar/1,baz/1]).
```

```

%% This is a type definition in Prolog syntax: declaration and code
:- true regtype bar(X) # "@var{X} is an acceptable kind of bar.".

bar(night).
bar(day).

%% This is another type definition in Prolog syntax, with no comment.
:- true regtype baz/1.

baz(a).
baz(b).

%% Two type definitions in 'typedef' syntax (will be expanded to code as above)
%% :- typedef aorb ::= ^a;^b.
%% :- typedef listof_or_aorb(X) ::= list(X);aorb.

%% Using functional notation:
:- regtype aorb/1.

aorb := a.
aorb := b.

%% Should use the other function syntax which uses *first argument* for return

:- regtype tree_of/2.

tree_of(_) := void.
tree_of(T) := tree(~regtype(T),~tree_of(T),~tree_of(T)).

%% tree_of(_, void).
%% tree_of(T, tree(X,L,R)) :-
%%     regtype(X,T),
%%     tree_of(T,L),
%%     tree_of(T,R).

:- regtype list_or_aorb/2.

list_or_aorb(T) := ~list(T).
list_or_aorb(_T) := ~aorb.

%% This is a property definition
%% This comment appears only in the place where the property is itself
%% is documented.
:- comment(long/1,"This is a property, describing a list that is longish.
    The definition is:

    @includedef{long/1}

    ").

```

```

%% The comment here will be used to document any predicate which has an
%% assertion which uses the property
:- prop long(L) # "@var{L} is rather long.".

long(L) :-
    length(L,N),
    N>100.

%% Now, a series of assertions:
%%
%% This declares the entry mode of this exported predicate (i.e.,
%% how it is called from outside).
:- entry p/3 : gnd * var * var.

%% This describes all the calls
:- calls p/3 : foo * bar * baz.

%% This describes the successes (for a given type of calls)
:- success p/3 : int * int * var => int * int * gnd.

%% This describes a global property (for a given type of calls)
:- comp p/3 : int * int * var + no_fail.

:- comment(p/3,"A @bf{general comment} on the predicate." ).
%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
    => int * int * list
    + (iso,no_fail)
    # "This mode is nice.".
:- pred p(Preds,Value,Assoc)
    : var * var * list
    => int * int * list
    + no_fail # "This mode is also nice.".
:- pred p/3
    => list * int * list
    + (no_fail,no_fail)
    # "Just playing around.".

:- pred q(A)
    : list(A)
    => (list(A),gnd(A))
    + no_fail
    # "Foo".
:- pred q(A)
    # "Not a bad use at all.".

:- pred q/2
    : var * {gnd,int}
    => {gnd,int} * int.
:- pred q/2

```

```

    :: int * list
    # "Non-moded types are best used this way.".

:- pred p/1 : var => list.

:- pred r(A)
    : list(A)
    => (list(A,int),gnd(A))
    + no_fail
    # "This uses parametric types".

:- comment(doinclude,s/1). %% Forces documentation even if not exported
:- pred s(A)
    : list(A)
    => (list(A),gnd(A))
    + no_fail.

:- comment(doinclude,[list/2,list/1]). %% Forces (local) documentation even if
                                         %% not exported

:- modedef og(A)
    => gnd(A)
    # "This is a @em{mode} definition: the output is ground.".

:- comment(doinclude,og/2).

:- modedef og(A,T)
    :: regtype(A,T)
    => gnd(A)
    # "This is a @em{parametric mode definition}.".

:- pred t(+A,-B,?C,@D,og(E))
    :: list * list * int * int * list
    : long(B)
    => (gnd(C),gnd(A))
    + no_fail
    # "This predicate uses @em{modes} extensively.".

%% Some other miscellaneous assertions:

%% Check is default assertion status anyway...
:- check pred u(+,-,og).
:- check pred u(int,list(mytype),int).

%% ‘‘true’’ status is normally compiler output
:- true pred w(+list(mytype)).

:- comment(doinclude,is/2).

:- trust pred is(Num,Expr) : arithexpression(Expr) => num(Num)
    # "Typical way to describe/document an external predicate (e.g.,

```

```
written in C).".

:- comment(doinclude,p/5).
:- pred p(og(int),in,@list(int),-,+A) + cost(1+length(A)).

%% Version information. The ciao.el emacs mode allows automatic maintenance

:- comment(version(0*1+2,1998/04/15,10:01*02+'MET DST'), "This comment
    includes the time. (Manuel Hermenegildo)").
:- comment(version(0*1+3,2001/1/2),"The next day is more boring.").
:- comment(version(0*1+1,2001/1/1),"A documentation odyssey!").

%% Control version comment prompting for the file.
%% Local Variables:
%% mode: CIAO
%% update-version-comments: "on"
%% End:
```


7 Auto Documenter Output for the Example Module

Author(s): Alan Robinson, David H.D. Warren.

Version: 0.1#3 (2001/1/2)

This is where general comments on the file go. In this case the file is a library which contains some assertion examples for testing the *automatic documentation system*.

7.1 Usage and interface (example_module)

- **Library usage:**
:- use_module(library(example_module)).
- **Exports:**
 - *Predicates:*
q/1, q/2, r/1, t/5, u/3, w/1, p/5.
 - *Properties:*
long/1.
 - *Regular Types:*
bar/1, baz/1, aorb/1, tree_of/2, list_or_aorb/2.
 - *Multifiles:*
p/3.
- **Other modules used:**
 - *Application modules:*
bar.
 - *Files of module user:*
foo.
 - *System library modules:*
engine/basic_props.
 - *Internal (engine) modules:*
arithmetic, atomic_basic, attributes, basic_props, basiccontrol, concurrency, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_basic, term_compare, term_typing.

7.2 Documentation on exports (example_module)

- bar/1:** REGTYPE
- Usage: bar(X)
- *Description:* X is an acceptable kind of bar.
 - *The following properties hold globally:*
Defines a regular type. (regtype/1)
- baz/1:** REGTYPE
- A regular type, defined as follows:


```
baz(a).
baz(b).
```

Usage:

- *The following properties hold globally:*

Defines a regular type.

(regtype/1)

aorb/1:

REGTYPE

A regular type, defined as follows:

```
aorb(a).
aorb(b).
```

Usage:

- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

tree_of/2:

REGTYPE

A regular type, defined as follows:

```
tree_of(_1,void).
tree_of(T,tree(_1,_2,_3)) :-
    regtype(T,_1),
    tree_of(T,_2),
    tree_of(T,_3).
```

Usage:

- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

list_or_aorb/2:

REGTYPE

A regular type, defined as follows:

```
list_or_aorb(T,_1) :-
    list(T,_1).
list_or_aorb(_T,_1) :-
    aorb(_1).
```

Usage:

- *The following properties should hold globally:*

Defines a regular type.

(regtype/1)

list/1:

PROPERTY

Usage 1: list(L)

- *Description:* L is a list.
- *The following properties hold globally:*

.(list(L),regtype(list(L)),[])

(undefined property)

Usage 2: list(L)

- *Description:* L is a list.
- *The following properties hold globally:*
Defines a regular type. (regtype/1)

Usage 3: list(L)

- *Description:* L is a list.
- *The following properties hold globally:*
Defines a regular type. (regtype/1)

q/1: PREDICATE**Usage 1:** q(A)

- *Description:* Foo
- *The following properties should hold at call time:*
A is a list. (list/1)
- *The following properties should hold upon exit:*
A is a list. (list/1)
A is ground. (gnd/1)
- *The following properties should hold globally:*
no_fail(q(A)) (undefined property)

Usage 2: q(A)

- *Description:* Not a bad use at all.

q/2: PREDICATE

The predicate is of type *dynamic*.

Usage 1:

- *The following properties should hold at call time:*
Arg1 is a free variable. (var/1)
Arg2 is ground. (gnd/1)
Arg2 is an integer. (int/1)
- *The following properties should hold upon exit:*
Arg1 is ground. (gnd/1)
Arg1 is an integer. (int/1)
Arg2 is an integer. (int/1)

Usage 2:

- *Description:* Non-moded types are best used this way.
- *Call and exit should be compatible with:*
Arg1 is an integer. (int/1)
Arg2 is a list. (list/1)

r/1:		PREDICATE
The predicate is of type <i>data</i> .		
Usage: <code>r(A)</code>		
– <i>Description:</i> This uses parametric types		
– <i>The following properties should hold at call time:</i>		
A is a list.		(list/1)
– <i>The following properties should hold upon exit:</i>		
A is a list of ints.		(list/2)
A is ground.		(gnd/1)
– <i>The following properties should hold globally:</i>		
<code>no_fail(r(A))</code>		(undefined property)
og/1:		PREDICATE
No further documentation available for this predicate.		
t/5:		PREDICATE
Usage: <code>t(+A,-(B),?(C),@(D),og(E))</code>		
– <i>Description:</i> This predicate uses <i>modes</i> extensively.		
– <i>Call and exit should be compatible with:</i>		
+A is a list.		(list/1)
-(B) is a list.		(list/1)
?(C) is an integer.		(int/1)
@(D) is an integer.		(int/1)
og(E) is a list.		(list/1)
– <i>The following properties should hold at call time:</i>		
B is rather long.		(long/1)
– <i>The following properties should hold upon exit:</i>		
C is ground.		(gnd/1)
A is ground.		(gnd/1)
– <i>The following properties should hold globally:</i>		
<code>no_fail(t(+A,-(B),?(C),@(D),og(E)))</code>		(undefined property)
u/3:		PREDICATE
Usage 1: <code>u(+,-,og)</code>		
Usage 2:		
– <i>Call and exit should be compatible with:</i>		
Arg1 is an integer.		(int/1)
Arg2 is a list of mytypes.		(list/2)
Arg3 is an integer.		(int/1)

w/1: PREDICATE
Usage: w(+list(mytype))

p/5: PREDICATE
Usage: p(og(int),in,@(list(int)),-,+A)
 – *The following properties should hold globally:*
 cost(p(og(int),in,@(list(int)),-,+A),1+length(A)) (undefined property)

long/1: PROPERTY
 This is a property, describing a list that is longish. The definition is:
 long(L) :-
 length(L,N),
 N>100.
Usage: long(L)
 – *Description:* L is rather long.

list/1: PROPERTY
Usage 1: list(L)
 – *Description:* L is a list.
 – *The following properties hold globally:*
 .(list(L),regtype(list(L)),[]) (undefined property)
Usage 2: list(L)
 – *Description:* L is a list.
 – *The following properties hold globally:*
 Defines a regular type. (regtype/1)
Usage 3: list(L)
 – *Description:* L is a list.
 – *The following properties hold globally:*
 Defines a regular type. (regtype/1)

7.3 Documentation on multifiles (example_module)

p/3: PREDICATE
 A **general comment** on the predicate.
 The predicate is *multifile* and of type *dynamic*.
General properties:
 – *If the following properties hold at call time:*
 Arg1 is ground. (gnd/1)
 Arg2 is a free variable. (var/1)
 Arg3 is a free variable. (var/1)

- *The following properties should hold at call time:*
`foo(Arg1)` (undefined property)
`Arg2` is an acceptable kind of bar. (bar/1)
`baz(Arg3)` (baz/1)
- *If the following properties hold at call time:*
`Arg1` is an integer. (int/1)
`Arg2` is an integer. (int/1)
`Arg3` is a free variable. (var/1)
then the following properties should hold upon exit:
`Arg1` is an integer. (int/1)
`Arg2` is an integer. (int/1)
`Arg3` is ground. (gnd/1)
- *If the following properties hold at call time:*
`Arg1` is an integer. (int/1)
`Arg2` is an integer. (int/1)
`Arg3` is a free variable. (var/1)
then the following properties should hold globally:
`no_fail(p(Arg1,Arg2,Arg3))` (undefined property)

Usage 1:

• ISO •

- *Description:* This mode is nice.
- *The following properties should hold at call time:*
`Arg1` is an integer. (int/1)
`Arg2` is an integer. (int/1)
`Arg3` is a free variable. (var/1)
- *The following properties should hold upon exit:*
`Arg1` is an integer. (int/1)
`Arg2` is an integer. (int/1)
`Arg3` is a list. (list/1)
- *The following properties should hold globally:*
Complies with the ISO-Prolog standard. (iso/1)
`no_fail(p(Arg1,Arg2,Arg3))` (undefined property)

Usage 2: p(Preds,Value,Assoc)

- *Description:* This mode is also nice.
- *The following properties should hold at call time:*
`Preds` is a free variable. (var/1)
`Value` is a free variable. (var/1)
`Assoc` is a list. (list/1)
- *The following properties should hold upon exit:*
`Preds` is an integer. (int/1)
`Value` is an integer. (int/1)
`Assoc` is a list. (list/1)
- *The following properties should hold globally:*
`no_fail(p(Preds,Value,Assoc))` (undefined property)

Usage 3:

- *Description:* Just playing around.
- *The following properties should hold upon exit:*
 - Arg1 is a list. (list/1)
 - Arg2 is an integer. (int/1)
 - Arg3 is a list. (list/1)
- *The following properties should hold globally:*
 - no_fail(p(Arg1,Arg2,Arg3)) (undefined property)
 - no_fail(p(Arg1,Arg2,Arg3)) (undefined property)

7.4 Documentation on internals (example_module)

s/1: PREDICATE

Usage: s(A)

- *The following properties should hold at call time:*
 - A is a list. (list/1)
- *The following properties should hold upon exit:*
 - A is a list. (list/1)
 - A is ground. (gnd/1)
- *The following properties should hold globally:*
 - no_fail(s(A)) (undefined property)

list/2: PROPERTY

Usage 1: list(L,T)

- *Description:* L is a list of Ts.
- *The following properties hold globally:*
 - .(list(L,T),regtype(list(L,T)),[]) (undefined property)

Usage 2: list(L,T)

- *Description:* L is a list of Ts.
- *The following properties hold globally:*
 - Defines a regular type. (regtype/1)

Usage 3: list(L,T)

- *Description:* L is a list of Ts.
- *The following properties hold globally:*
 - Defines a regular type. (regtype/1)

og/2: MODE

Usage: og(A,T)

- *Description:* This is a *parametric mode definition*.
- *Call and exit are compatible with:*
 - regtype(A,T) (undefined property)
- *The following properties are added upon exit:*
 - A is ground. (gnd/1)

is/2:

PREDICATE

Usage: Num is Expr

- *Description:* Typical way to describe/document an external predicate (e.g., written in C).
- *The following properties should hold at call time:*
Expr is an arithmetic expression. (arithexpression/1)
- *The following properties hold upon exit:*
Num is a number. (num/1)

8 Installing lpd doc

lpd doc can be obtained as either a source or binary distribution. The source distribution contains all the source code and libraries and can be compiled on a supported Prolog system (lpd doc is developed using CIAO Prolog). The latest publically distributed version of lpd doc is available from <http://www.clip.dia.fi.upm.es/Software>. A newer version in Beta test is often available in <http://www.clip.dia.fi.upm.es/Software/Beta>.

8.1 Installing the source distribution

- Uncompress (using `gunzip`) and unpackage (using `tar -xpf`) the distribution in a suitable directory. This will create a new directory called `lpd doc` as well as a link `lpd doc-X.Y` to this directory, where `X.Y` is the version number of the distribution. The `-p` option in the `tar` command ensures that the relative dates of the files in the package are preserved, which is needed for correct operation of the Makefiles.
- Enter the newly created directory, and edit the file `SETTINGS` in a text editor (edit the one in that directory, not the ones in the subdirectories).
 - Decide which Prolog/CLP system you will use for compiling lpd doc (actually, currently only CIAO is supported – but porting to, e.g., SICStus Prolog should not be too difficult) and modify the first part of the `SETTINGS` file accordingly. The `DOCDIR` directory should not be an existing `info` directory, since this will overwrite the `dir` file in that directory.
 - Select the directories in which you want the lpd doc binaries, libraries, and documents installed, by setting the corresponding variables in `SETTINGS`.
- Type `gmake install`. This should create the lpd doc executable and install it in the `BINDIR` directory, install the lpd doc library in a separate directory in the `LIBDIR` directory, and install the lpd doc documentation in the `DOCDIR` directory.
- In order for the lpd doc documentation to be available to users automatically, certain environment variables have to be set. The installation leaves files suitable for inclusion in initialization scripts (e.g., `DOTcshrc` for `csh`) in the lpd doc library.

8.2 Installing the binary distribution

If you do not have one of the Prolog systems recommended for compiling lpd doc (or you simply prefer to use a pre-compiled version) you can still install the binary distribution as follows:

- If you have not already done so, download from the same distribution site as lpd doc (e.g., <http://www.clip.dia.fi.upm.es/Software>) the CIAO engine executable for your architecture/OS (e.g., SPARC/SunOS, SPARC/Solaris, x86/Linux, x86/Windows, etc.). Install it and set the appropriate environment variables as indicated in the installation instructions.
- Follow the same steps for lpd doc installation as above, up to but not including `gmake install`, and skipping the step related to selecting under which Prolog system to compile.
- Download from the lpd doc distribution site the lpd doc CIAO binary, and place it in the `lpd doc/src` directory. Change the date (e.g., type “ `touch lpd doc`”) so that it will not be recompiled.
- Go back one level to the root lpd doc directory and type `gmake install`.
- Continue with the installation as above.

8.3 Other software packages required

The basic functionality of `lpdoc` (generating the `.texi`, the `.man1`, the `index` files) is essentially self contained. However, using the full capabilities of `lpdoc` requires having several other software packages installed. Fortunately, all of these packages are public domain software and they will normally be already installed in, e.g., a standard Linux distribution.

- **Basic requirements:** the Makefiles used by `lpdoc` require GNU Make (`gmake`), and for now have only been designed for UN*X-like operating systems.
- **Generating .dvi files:** `lpdoc` normally generates `.texi` files (actually, a number of `.texic` files). From the `.texi` files, `.dvi` files are generated using the standard `tex` package directly. This requires that the `texinfo.tex` file (containing the relevant macros) be available to `tex`. This file is normally included with modern `tex` distributions, and is also provided in the `lpdoc` library for convenience. The `.dvi` files can also be generated with the GNU `Texinfo` package, which provides, among others, the `texi2dvi` command. However, `Texinfo` itself requires the standard `tex` document processing package. In order to use `texi2dvi` instead of `tex` when processing documents you should change the variable `TEX` in the `Makefile.skel` file in the `lib` directory before installing `lpdoc`.

The `texindex` package is required in order to process the indices. If you use references in your manual, then the `bibtex` package is also needed. `texindex` and `bibtex` are included with most `tex` distributions.

- **Generating .ps files:** `.ps` files are generated from the `.dvi` files using the `dvips` command (this, again, can be changed in the `Makefile.skel` file in the `lib` directory). This command is included with standard `tex` distributions.
- **Generating .pdf files:** `.pdf` files are currently generated from the `.texi` file using the `pdftex` command (this, again, can be changed in the `Makefile.skel` file in the `lib` directory). This command is included in current Linux distributions.
- **Generating .html files:** `.html` files are generated directly from the `.texi` file using the `texi2html` command (this, again, can be changed in the `Makefile.skel` file in the `lib` directory). This command is a `perl` script and is included with the `lpdoc` distribution, and installed in the library (so that it does not overwrite other existing versions). It is also typically included in the `Texinfo` distribution. A required intermediate step is to resolve the link references which appear in the `.texi` file (the `.texi` file includes all the `.texic` files and has all references resolved). This is done using the `emacs` editor in batch mode, calling functions in the `emacs-library.el` file included in the `lib` directory of the `lpdoc` distribution. Thus, a recent version of `emacs` is required for this purpose.
- **Generating .info files:** `.info` files are also generated directly from the `.texi` file using the `makeinfo` command (this, again, can be changed in the `Makefile.skel` file in the `lib` directory). This command is included in the `Texinfo` distribution. Resolving the link references in the `.texi` file is also required as above.
- If pictures are used in the manual, and `html` output is selected, the commands `pstogif` and `cjpeg` are also required, in order to convert the figures from `.eps` to `.jpg` format.

References

- [Bue95] F. Bueno.
The CIAO Multiparadigm Compiler: A User's Manual.
Technical Report CLIP8/95.0, Facultad de Inform\`atica, UPM, June 1995.
- [Bue98] F. Bueno.
Using Assertions for Static Debugging of CLP: A Manual.
Technical Report CLIP1/98.0, DISCIPL Project/CLIP Group, UPM, June 1998.
- [DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni.
Prolog: The Standard.
Springer-Verlag, 1996.
- [Her99] M. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
Technical Report CLIP10/99.0, Facultad de Inform\`atica, UPM, September 1999.
- [JM94] J. Jaffar and M.J. Maher.
Constraint Logic Programming: A Survey.
Journal of Logic Programming, 19/20:503–581, 1994.
- [Knu84] D. Knuth.
Literate programming.
Computer Journal, 27:97–111, 1984.
- [PBH97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
Technical Report CLIP2/97.1, Facultad de Inform\`atica, UPM, July 1997.
- [PBH98] G. Puebla, F. Bueno, and M. Hermenegildo.
A Framework for Assertion-based Debugging in Constraint Logic Programming.
In *Proceedings of the JICSLP'98 Workshop on Types for CLP*, pages 3–15, Manchester, UK, June 1998.

Predicate/Method Definition Index

C

check/1 37

F

false/1 38

I

is/2 58

O

og/1 54

P

p/3 55

p/5 55

Q

q/1 53

q/2 53

R

r/1 53

S

s/1 57

T

t/5 54

true/1 37

trust/1 37

U

u/3 54

W

w/1 54

Property Definition Index

D

`docstring/1` 21, 44

H

`head_pattern/1` 40

L

`list/1` 52, 55

`list/2` 57

`long/1` 55

S

`stringcommand/1` 22

Regular Type Definition Index

A

aorb/1	52
assrt_body/1	39
assrt_status/1	43
assrt_type/1	44

B

bar/1	51
baz/1	51

C

c_assrt_body/1	42
complex_arg_property/1	40
complex_goal_property/1	41

F

filetype/1	27
------------------	----

G

g_assrt_body/1	43
----------------------	----

L

list_or_aorb/2	52
----------------------	----

P

predfunctor/1	44
property_conjunction/1	41
property_starterm/1	41
propfunctor/1	44

S

s_assrt_body/1	42
----------------------	----

T

time_struct/1	28
tree_of/2	52

V

version_descriptor/1	27
version_maintenance_type/1	28
version_number/1	28

Y

ynd_date/1	28
------------------	----

Mode Definition Index

O	og/2	57
---	------------	----

Concept Definition Index

•		@includefact command	27
.bib files	12, 23	@includeverbatim command	26
@		@index command	22
@! command	26	@iso command	26
@' command	25	@item command	24
@. command	25	@j command	26
@.. command	25	@key command	25
@= command	25	@l command	26
@? command	26	@L command	26
@' command	25	@lib command	23
@~ command	25	@noindent command	25
@~ command	25	@o command	26
@aa command	26	@O command	26
@AA command	26	@oe command	26
@ae command	26	@OE command	26
@AE command	26	@op command	23
@apl command	23	@p command	25
@b command	26	@pred command	23
@begin{cartouche} command	25	@ref command	24
@begin{description} command	24	@result command	26
@begin{enumerate} command	24	@section command	25
@begin{itemize} command	24	@sp command	25
@begin{verbatim} command	25	@ss command	26
@bf command	25	@subsection command	25
@bullet command	26	@t command	26
@c command	26	@today command	25
@cindex command	22	@tt command	25
@cite command	23	@u command	26
@comment command	24	@uref command	24
@concept command	22	@v command	26
@copyright command	26	@var command	23
@d command	26		
@decl command	23	A	
@em command	25	accents	25
@email command	24	acceptable modes	40
@end{cartouche} command	25	application	3
@end{description} command	24	assertion body syntax	39, 42, 43
@end{enumerate} command	24	assertions	21
@end{itemize} command	24	avoiding indentation	25
@end{verbatim} command	25		
@file command	23	B	
@footnote command	25	bibliographic citations	23
@H command	26	bibliographic entries	12
@hfill command	25	bibliographic entry	23
@i command	26	bibtex	12, 23
@image command	27	blank lines	25
@include command	26	bold face	25
@includedef command	27		

C

calls assertions	33
check assertions	37
comment	24
comment assertions	37
comments, machine readable	31
comp assertions	33, 34
component files	4, 11
contents area	13

D

date	25
decl assertions	36
description list	24
documentation strings	21

E

emacs, accessing info files	14
email address	24
email addresses	23
emphasis face	25
encapsulated postscript	27
entry assertions	35
enumerated list	24
escape sequences	22
example of lpd doc use	45

F

false assertions	38
fixed format text	25
fixed-width font	25
footnote	25
formatting commands	22, 31
framed box	25

H

html index page	13
-----------------------	----

I

image file	27
images, inserting	27
images, scaling	27
including a predicate definition	27
including an image	27
including code	26

including files	26
including images	26
indentation, avoiding	25
info path list	15
installation	3
installation, of manuals	13
internals manual	16
italics face	25
item in an itemized list	24
itemized list	24

K

keyboard key	25
--------------------	----

L

library	3
literate programming	14

M

machine readable comments	21
main file	4, 11
Makefile	11, 60

N

new item in description list	24
------------------------------------	----

P

page numbering, changing	12
paragraph break	25
parts in a large document	17
pred assertions	32, 33
prop assertions	34

R

references	23
regtype assertions	34, 35

S

section	25
sections	23
SETTINGS	11
sharing pieces of text	27
space, extra lines	25
spcae, horizontal fill	25
special characters	25
strong face	25
subsection	25
success assertions	33
synopsis section of the man page	12
syntax of formatting commands	22
system modules	11

T

texinfo files	4
textual comments	21

true assertions	38
trust assertions	37
typewriter-like font	25

U

Universal Resource Locator	24
URL	24
urls	23
usage of a command	27
usage of the application	12

V

verbatim text	25
---------------------	----

W

WWW address	24
-------------------	----

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

*		<code>@decl</code> command.....	23
<code>*/2</code>	41	<code>@em</code> command.....	25
.		<code>@email</code> command.....	24
<code>.bib</code> files.....	12, 23	<code>@end{cartouche}</code> command.....	25
:		<code>@end{description}</code> command.....	24
<code>::/2</code>	32	<code>@end{enumerate}</code> command.....	24
=		<code>@end{itemize}</code> command.....	24
<code>=>/2</code>	32	<code>@end{verbatim}</code> command.....	25
@		<code>@file</code> command.....	23
<code>@!</code> command.....	26	<code>@footnote</code> command.....	25
<code>@'</code> command.....	25	<code>@H</code> command.....	26
<code>@.</code> command.....	25	<code>@hfill</code> command.....	25
<code>@..</code> command.....	25	<code>@i</code> command.....	26
<code>@=</code> command.....	25	<code>@image</code> command.....	27
<code>@?</code> command.....	26	<code>@include</code> command.....	26
<code>@'</code> command.....	25	<code>@includedef</code> command.....	27
<code>@~</code> command.....	25	<code>@includefact</code> command.....	27
<code>@^</code> command.....	25	<code>@includeverbatim</code> command.....	26
<code>@aa</code> command.....	26	<code>@index</code> command.....	22
<code>@AA</code> command.....	26	<code>@iso</code> command.....	26
<code>@ae</code> command.....	26	<code>@item</code> command.....	24
<code>@AE</code> command.....	26	<code>@j</code> command.....	26
<code>@apl</code> command.....	23	<code>@key</code> command.....	25
<code>@b</code> command.....	26	<code>@l</code> command.....	26
<code>@begin{cartouche}</code> command.....	25	<code>@L</code> command.....	26
<code>@begin{description}</code> command.....	24	<code>@lib</code> command.....	23
<code>@begin{enumerate}</code> command.....	24	<code>@noindent</code> command.....	25
<code>@begin{itemize}</code> command.....	24	<code>@o</code> command.....	26
<code>@begin{verbatim}</code> command.....	25	<code>@O</code> command.....	26
<code>@bf</code> command.....	25	<code>@oe</code> command.....	26
<code>@bullet</code> command.....	26	<code>@OE</code> command.....	26
<code>@c</code> command.....	26	<code>@op</code> command.....	23
<code>@cindex</code> command.....	22	<code>@p</code> command.....	25
<code>@cite</code> command.....	12, 23	<code>@pred</code> command.....	23
<code>@comment</code> command.....	24	<code>@ref</code> command.....	24
<code>@concept</code> command.....	22	<code>@result</code> command.....	26
<code>@copyright</code> command.....	26	<code>@section</code> command.....	25
<code>@d</code> command.....	26	<code>@sp</code> command.....	25
		<code>@ss</code> command.....	26
		<code>@subsection</code> command.....	25
		<code>@t</code> command.....	26
		<code>@today</code> command.....	25
		<code>@tt</code> command.....	25
		<code>@u</code> command.....	26
		<code>@uref</code> command.....	24
		<code>@v</code> command.....	26

@var command..... 23

+

+/1 40

A

accents 25
 acceptable modes 40
 address 24
 analyzer output 38
 aorb/1 51, 52
 application 3
 arithmetic 21, 32, 39, 51
 assertion body syntax 39, 42, 43
 assertions 3, 18, 21, 31, 32, 39
 assertions/assrt_props 32
 assrt_body/1 32, 39
 assrt_status/1 39, 43
 assrt_type/1 39, 44
 atomic_basic 21, 32, 39, 51
 attributes 21, 32, 39, 51
 autodoc 3
 autodocformats 3, 11, 12
 automatic documentation library 3
 avoiding indentation 25

B

bar 51
 bar/1 51
 basic_props 21, 32, 39, 51
 basiccontrol 21, 32, 39, 51
 basicprops 3
 baz/1 51
 bibliographic citations 23
 bibliographic entries 12
 bibliographic entry 23
 bibtex 8, 12, 23, 60
 blank lines 25
 bold face 25

C

c_assrt_body/1 39, 42
 c_itf 7
 call/1 42
 calls assertions 33
 calls/1 32, 33, 35
 calls/2 32, 33
 character string 31
 check assertions 37
 check/1 32, 37, 38
 CIAO 3, 14, 15, 45, 59
 ciaoc 6, 7
 cjpeg 60
 CLP 3
 comment 24
 comment assertions 37
 comment string 40, 42, 43
 comment/2 7, 13, 17, 18, 21, 22, 32, 37
 comments 3, 18
 comments, machine readable 31
 comment/2 8
 comp assertions 33, 34
 comp/1 32, 33, 34, 43
 comp/2 32, 34
 compatible 39
 complex argument property 39, 40, 42, 43
 complex goal property 40, 42, 43
 complex_arg_property/1 39, 40, 42, 43
 complex_goal_property/1 39, 40, 41, 43
 component files 4, 11
 concurrency 21, 32, 39, 51
 contents area 13
 conversion formats 3
 csh 15, 59

D

data_facts 21, 32, 39, 51
 date 25
 dcg_expansion 21, 39
 decl assertions 36
 decl/1 32, 36
 decl/2 32, 36
 description list 24
 dir 14, 59
 docstring/1 21, 31, 39, 40, 42, 43, 44

documentation strings 21
 DOTcshrc 59
 dvips 60
 dynamic/1 13

E

emacs 8, 13, 14, 15, 29, 60
 emacs, accessing info files 14
 emacs-library.el 60
 email address 24
 email addresses 8, 23
 emphasis face 25
 encapsulated postscript 27
 engine(basic_props) 35
 engine/basic_props 51
 ensure_loaded/1 7, 17
 entry assertions 35
 entry/1 32, 35, 36, 42
 enumerated list 24
 environment variables 59
 escape sequences 22
 example of lpdoc use 45
 exceptions 21, 32, 39, 51

F

false assertions 38
 false/1 32, 38
 filetype/1 21, 27
 fixed format text 25
 fixed-width font 25
 foo 51
 footnote 25
 formatting commands 3, 21, 22, 31
 framed box 25
 func/1 39, 42

G

g_assrt_body/1 39, 43
 gmake 60
 gmake all 11, 12, 14
 GNU general public license 1
 GNU Make 60
 gunzip 59

H

head pattern 39, 40, 43
 head_pattern/1 39, 40, 43
 html 4
 html index page 13

I

image file 27
 images, inserting 27
 images, scaling 27
 include files 17
 include/1 17
 including a predicate definition 27
 including an image 27
 including code 26
 including files 26
 including images 26
 indentation, avoiding 25
 index_comment/2 12
 info 1, 4, 14, 15, 16, 19, 25, 59
 info path list 15
 inserting images 8
 INSTALL.lpdoc 18
 installation 3
 installation, of manuals 13
 integer/1 41
 internals manual 3, 16, 17
 io_aux 21, 32, 39, 51
 io_basic 21, 32, 39, 51
 is/2 58
 iso/1 7
 italics face 25
 item in an itemized list 24
 itemized list 24

K

keyboard key 25

L

LaTeX	22
library	3
library(modes)	40
Linux	60
list/1	52, 55
list/2	41, 57
list_or_aorb/2	51, 52
literate programming	14
long/1	51, 55
lpdoc 1, 3, 4, 6, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 24, 31, 37, 40, 44, 45, 59, 60	
lpdoc -help	12
lpdoc library directory	11

M

machine readable comments	21
main file	4, 11
main/0	3
main/1	3
Makefile	4, 11, 12, 13, 60
makeinfo	60
man	14, 15
master index	4
meta_predicate/1	13
mode	32, 40
modedef/1	32, 36, 40

N

n_assrt_body/5	42, 43
netscape	6, 14
new item in description list	24
not_further_inst/1	42

O

og/1	54
og/2	57
op/3	13
option_comment/2	11

P

p/3	51, 55
p/5	51, 55
packages	17
page numbering, changing	12
paragraph break	25
parts in a large document	17
pdftex	60
perl	60
pred assertions	32, 33
pred/1	23, 32, 33, 36, 39, 42
pred/2	32
predfunctor/1	39, 44
predname/1	40
program assertions	31
Prolog	3, 11, 17, 59
Prolog source files	3
prolog_flags	21, 32, 39, 51
prop assertions	34
prop/1	32, 34
prop/2	32, 34
properties	3
property	34
property_conjunction/1	39, 40, 41
property_starterm/1	39, 40, 41
propfunctor/1	39, 44
providing information to the compiler	36, 37
pstogif	60

Q

q/1	51, 53
q/2	51, 53

R

r/1	51, 53
references	23, 60
regtype assertions	34, 35
regtype/1	32, 34, 35
regtype/2	32, 35
regular type	34

S

s/1 57
 s_assrt_body/1 39, 42
 scribe 22
 section 25
 sections 23
 SETTINGS 4, 6, 11, 12, 13, 14, 15, 16, 17, 18, 24
 sharing pieces of text 27
 space, extra lines 25
 spcae, horizontal fill 25
 special characters 25
 specifications 31
 streams_basic 21, 32, 39, 51
 stringcommand/1 21, 22, 37, 40, 42, 43, 44
 strong face 25
 subsection 25
 success assertions 33
 success/1 32, 33
 success/2 32, 33
 synopsis section of the man page 12
 syntax of formatting commands 22
 system modules 11
 system_info 21, 32, 39, 51

T

t/5 51, 54
 tar 59
 term_basic 21, 32, 39, 51
 term_compare 21, 32, 39, 51
 term_typing 21, 32, 39, 51
 tex 19, 60
 TeX 7
 texi2dvi 60
 texi2html 60
 texindex 60
 texinfo 1, 3, 4, 12, 16, 19
 Texinfo 60
 texinfo files 4
 textual comments 21
 time_struct/1 28
 touch 59
 tree_of/2 51, 52
 troubleshooting 11
 true assertions 38
 true/1 32, 37

Y

ymd_date/1 28

trust assertions 37
 trust/1 32, 37
 type/1 32, 35
 type/2 32, 35
 types 3
 typewriter-like font 25

U

u/3 51, 54
 Universal Resource Locator 24
 unix 15
 URL 24
 url references 8
 urls 23
 usage 32
 usage of a command 27
 usage of the application 12
 usage section 8
 usage tips 11
 usage_message/1 8, 12
 use_module/1 18
 use_package/1 17
 using citations 12

V

var/1 41
 variable names 31
 verbatim text 25
 version_descriptor/1 21, 27
 version_maintenance_type/1 28
 version_number/1 28

W

w/1 51, 54
 word-help 15
 word-help-setup.el 15
 word-help.el 15
 WWW 1, 14
 WWW address 24

X

xdvi 12

