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: 23 April 2002

Version 1.9#58 (2002/4/19, 20:59:33 CEST)

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

Table of Contents

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.

Note: `lpdoc` is currently fully supported only on Linux and other Un*x-like systems, due to the use of `Makefiles` and other Un*x-related utilities. It is possible to run `lpdoc` under Win32 using `Cygwin`. A version which is written entirely in Prolog and will thus run standalone also on Win32 is currently under beta testing.

This documentation corresponds to version 1.9#58 (2002/4/19, 20:59:33 CEST).

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 12 [Installing lpdoc], page 83.

Other parts of this document provide:

- Documentation on the syntax and meaning of the *assertions* that `lpdoc` uses (those defined in the `Ciao` `assertions` library [PBH97,PBH98,Bue98]). These include *comment* assertions (containing basically documentation text), formal assertions (containing properties), and combined assertions.
- Documentation on a basic set of properties, types, etc. which are predefined in the `Ciao` `basic_props`, `regtypes`, `native_props`, and `meta_props` libraries. These properties, and any others defined by the user or in other `Ciao` libraries, can be used in program assertions.
- 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`.

Some additional information on `lpdoc` can be found in [Her00].

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 2.6 [Enhancing the documentation being generated], page 16). 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 2.8 [Some usage tips], page 19).

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 2.6 [Enhancing the documentation being generated], page 16).

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., `manl` 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 2.5 [Installing a generated manual in a public area], page 15).

1.3 lpdoc 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 2.2 [Generating a manual], page 13), 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 ]
        [-p start_page ]
        [-t paper_type ]
        -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
  such 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. -u indicates a Prolog file with defi-
  nitions to be loaded.

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.
- `@`foonote 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 `ciaoc` 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.
- Nicer 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 correspoding 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 `commment/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 lpdoc can read this info easily. Using this guarantees that the version numbers of the manuals always concide 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 chapeter, 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. `Makefile`s now also install documentation in public areas and produce global indices. Several documents can cohexist 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 (-nobugs -noauthors -noversion -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.

PART I - LPdoc Reference Manual

2 Generating Installing and Accessing Manuals

Author(s): Manuel Hermenegildo.

Version: 1.9#58 (2002/4/19, 20:59:33 CEST)

Version of last change: 1.9#57 (2002/1/5, 0:47:15 CET)

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 from the Ciao emacs mode

If you use the `emacs` editor (highly recommended in all circumstances), then the simplest way to quickly generate a manual is by doing it from the Ciao emacs mode (this mode comes with the Ciao Prolog distribution and is automatically installed with Ciao). The Ciao emacs mode provides menu- and keyboard-binding driven facilities for generating a stand-alone document with the documentation corresponding to the file in the buffer being visited by emacs. This is specially useful while modifying the source of a file, in order to check the output that will be produced when incorporating this file into a larger document. It is also possible to generate more complex documents, by editing the (automatically provided) *SETTINGS* in the same way as when generating a manual from the command line (see below). However, when generating complex documents, it is best to devote an independent, permanent directory to the manual, and the full procedure described in the rest of this text is preferred.

2.2 Generating a manual

The `lpdoc` library directory includes two generic files which are quite useful for the generation of complete manuals: the *Makefile* and *SETTINGS* files. 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 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 12.1 [Installing the source distribution], page 83).
- 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., including bug information , versions and patches or only patches , authors , changelog , explanation of modes, *one-sided printing* (*two-sided* is the default), 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`, `manl`, ...) 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 `manl` 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.
- `PAPERTYPE` (default value `afourpaper`) allows select several paper sizes for the printable outputs (`dvi`, `ps`, etc.). The currently supported outputs (most of them inherited from `texinfo`) are:

  `afourpaper`
  : The default, usable for printing on *A4 paper*. Rather busy, but saves trees.

  `afourwide`
  : This one crams even more stuff than `afourpaper` on an A4 page. Useful for generating manuals in the least amount of space. Saves more trees.

  `afourlatex`
  : This one is a little less compressed than `afourpaper`.

  `smallbook`
  : Small pages, like in a handbook.

`letterpaper`
: For printing on American *letter size paper.*

`afourthesis`
: A *thesis-like style* (i.e., double spaced, wide margins etc.). Useful – for inserting `lpdoc` output as appendices of a thesis or similar document. Does not save trees.

- Type `gmake all` (you can also type `gmake dvi`, `gmake html`, `gmake ps` or `gmake info`, ... to force generation of a particular target).

2.3 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 ⟨R⟩ in the `xdvi` window. The other formats can be generated later, once the `.dvi` version has the desired contents.

2.4 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 12.2 [Other software packages required], page 83), 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 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.

2.5 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. A variable EXTRAFILES allows defining a list of additional files which will be copied to the DOCDIR installation directory. This is useful to place figures or other files which the HTML header files use in the installation directory (so that paths can be local). These files must reside in the directory in which the documentation is being generated.

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.

Important Note: In order for the different components to appear in the correct positions in the index pages mentioned above the traditional ('C') Lexical order must be active. In recent Un*x systems (e.g., in most current Linux systems) this may not be the case. There are several possible fixes:

- For csh put setenv LC_COLLATE C in your .cshrc.
- For bash put export LC_COLLATE=C in your .profile.
- In many systems this can be done globally by the super-user. E.g., in many Linux systems set LANG="C" in /etc/sysconfig/i18n.

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.

2.6 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 `lpdoc` library). Using this library, such assertions and comments are simply ignored by the compiler. This compatibility library also allows compiling `lpdoc` itself under (C)LP systems other than the `Ciao` system under which it is developed.

2.7 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.

2.7.1 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 publicly readable on the WWW, they should be copied into a directory visible by browsers running in other machines, such as `/home/clip/public_html/lpdoc_docs`, `/usr/home/httpd/htmldocs/lpdoc_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`.

2.7.2 Accessing info manuals

Generated `.info` files are meant to be viewed by the `emacs` editor or by the standalone `info` application, both publicly 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 `lpdoc` (`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 publicly 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 `lpdoc`, 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`).

2.7.3 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/manl`) 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 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`.

2.7.4 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/lpdoc_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 `lpdoc` library.

More complex setups can be accommodated, 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.

2.8 Some usage tips

This section contains additional suggestions on the use of `lpdoc`.

2.8.1 Ensuring Compatibility with All Supported Target Formats

One of the nice things about `lpdoc` 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 `lpdoc` 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 "lpdoc 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.

2.8.2 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 `lpdoc` 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 `lpdoc` 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*).

2.8.3 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.

2.8.4 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 (`:- comment(filetype,...).`) 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`). Adding this declaration will, for example, avoid spurious documentation of the declarations in the `assertions` package themselves when this package is included in a user file.

2.8.5 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.

2.8.6 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.

2.8.7 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.lpdoc}").
```

will include the contents of the file `Description.lpdoc` 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 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`.

2.8.8 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.lpdoc` 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.lpdoc}
   ...
   ").
```

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.lpdoc}").
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`.

2.9 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 installed version of `tex`/ `texinfo` installations, as mentioned in Section 2.2 [Generating a manual], page 13. 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#58 (2002/4/19, 20:59:33 CEST)

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:*

    `strings`.
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `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`)

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 on of a number of `bibtex` files (*.bib files*) . A reference in brackets ([ ]) is inserted in the text an 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 commads 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.

  `@begin{verbatim}`
  : marks the beginning of *fixed format text*, such as a program example. A fixed-width, typewriter-like font is used.

  `@end{verbatim}`
  : marks the end of formatted text.

`@begin{cartouche}`
: marks the beginning of a section of text in a *framed box*, with round corners.

`@end{cartouche}`
: marks the end of a section of text in a framed box.

`@section{`*text*`}`
: starts a *section* whose title is *text*. Due to a limitation of the `info` format, do not use `:` or `-` or `,` in section, subsection, title (chapter), etc. headings.

`@subsection{`*text*`}`
: starts a *subsection* whose title is *text*.

`@footnote{`*text*`}`
: places *text* in a *footnote*.

`@today`
: prints the current *date*.

`@hfill`
: introduces horizontal filling space (may be ignored in certain formats).

`@bf{`*text*`}`
: *text* will be formatted in *bold face* or any other *strong face*.

`@em{`*text*`}`
: *text* will be formatted in *italics face* or any other *emphasis face*.

`@tt{`*text*`}`
: *text* will be formatted in a *fixed-width font* (i.e., *typewriter-like font*).

`@key{`*key*`}`
: *key* is the identifier of a *keyboard key* (i.e., a letter such as `a`, or a special key identifier such as `RET` or `DEL`) and will be formatted as ⟨LFD⟩ or in a fixed-width, typewriter-like font.

`@sp{`*N*`}`
: generates *N* *blank lines* of space. Forces also a paragraph break.

`@p`
: forces a *paragraph break*, in the same way as leaving one or more blank lines.

`@noindent`
: used at the beginning of a paragraph, states that the first line of the paragraph should not be indented. Useful, for example, for *avoiding indentation* on paragraphs that are continuations of other paragraphs, such as after a verbatim.

- **Accents and special characters:**

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

`@‘{o}`	⇒ ò
`@’{o}`	⇒ ó
`@^{o}`	⇒ ô
`@..{o}`	⇒ ö
`@~{o}`	⇒ õ
`@={o}`	⇒ ō
`@.{o}`	⇒ ȯ
`@u{o}`	⇒ ŏ
`@v{o}`	⇒ ǒ
`@H{o}`	⇒ ő
`@t{oo}`	⇒ o͡o

`@c{o}`	⇒ o̧
`@d{o}`	⇒ ọ
`@b{o}`	⇒ o̱
`@oe`	⇒ œ
`@OE`	⇒ Œ
`@ae`	⇒ æ
`@AE`	⇒ Æ
`@aa`	⇒ å
`@AA`	⇒ Å
`@o`	⇒ ø
`@O`	⇒ Ø
`@l`	⇒ ł
`@L`	⇒ Ł
`@ss`	⇒ ß
`@?`	⇒ ¿
`@!`	⇒ ¡
`@i`	⇒ ı
`@j`	⇒ ȷ
`@copyright`	⇒ ©
`@iso`	⇒ ⟨● 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(CO)`

– *Description:* `CO` 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.

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.

3.3 Documentation on internals (`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).

Usage 1: :- `comment(CommentType,TitleText).`

- *Description:* Provides a *title* for the module, library, or application. When generating documentation automatically, the text in `TitleText` will be used appropriately (e.g., in the cover page as document title or as chapter title if part of a larger document). This will also be used as a brief description of the manual in on-line indices. There should be at most one of these declarations per module.
- *Example:*

```
:- comment(title,"Documentation-Oriented Assertions").
```

Usage 2: :- `comment(CommentType,SubTitleText)`.

- *Description:* Provides *subtitle* lines. This can be, e.g., an explanation of the application to add to the title, the address of the author(s) of the application, etc. When generating documentation automatically, the text in `SubTitleText` will be used accordingly. Several of these declarations can appear per module, which is useful for, e.g., multiple line addresses.
- *Example:*

```
:- comment(subtitle,"A Reference Manual").
```

- *The following properties should hold upon exit:*

  `CommentType` and `subtitle` unify. (`= /2`)

  `SubTitleText` is a *documentation string.* (`docstring/1`)

Usage 3: :- `comment(CommentType,AuthorText)`.

- *Description:* Provides the *author*(s) of the module or application. If present, when generating documentation for the module automatically, the text in `AuthorText` will be placed in the corresponding chapter or front page. There can be more than one of these declarations per module. In order for author indexing to work properly, please use one author declaration per author. If more explanation is needed (who did what when, etc.) use an acknowledgements comment.
- *Example:*

```
:- comment(author,"Alan Robinson").
```

- *The following properties should hold upon exit:*

  `CommentType` and `author` unify. (`= /2`)

  `AuthorText` is a *documentation string.* (`docstring/1`)

Usage 4: :- `comment(CommentType,AckText)`.

- *Description:* Provides *acknowledgements* for the module. If present, when generating documentation for the module automatically, the text in `AckText` will be placed in the corresponding chapter or section. There can be only one of these declarations per module.
- *Example:*

```
:- comment(ack,"Module was written by Alan, but others helped.").
```

- *The following properties should hold upon exit:*

  `CommentType` and `ack` unify. (`= /2`)

  `AckText` is a *documentation string.* (`docstring/1`)

Usage 5: :- `comment(CommentType,CopyrightText)`.

- *Description:* Provides a *copyright* text. This normally appears somewhere towards the beginning of a printed manual. There should be at most one of these declarations per module.
- *Example:*

```
:- comment(copyright,"Copyright © 2001 FSF.").
```

- *The following properties should hold upon exit:*

  `CommentType` and `copyright` unify. (`= /2`)

  `CopyrightText` is a *documentation string.* (`docstring/1`)

Usage 6: :- `comment(CommentType,SummaryText)`.

- *Description:* Provides a brief global explanation of the application or library. The text in `SummaryText` will be used as the *abstract* for the whole manual. There should be at most one of these declarations per module.
- *Example:*

```
:- comment(summary,"This is a @bf{very} important library.").
```

- *The following properties should hold upon exit:*

  `CommentType` and `summary` unify. (`= /2`)

  `SummaryText` is a *documentation string.* (`docstring/1`)

Usage 7: :- `comment(CommentType,CommentText)`.

- *Description:* Provides the main comment text for the module or application. When generating documentation automatically, the text in `CommentText` will be used as the *introduction* or *main body* of the corresponding chapter or manual. There should be at most one of these declarations per module. `CommentText` may use subsections if substructure is needed.
- *Example:*

```
:- comment(module,"This module is the @lib{comments} library.").
```

- *The following properties should hold upon exit:*

  `CommentType` and `module` unify. (`= /2`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 8: :- `comment(CommentType,CommentText)`.

- *Description:* Provides additional comments text for a module or application. When generating documentation automatically, the text in `CommentText` will be used in one of the last sections or appendices of the corresponding chapter or manual. There should be at most one of these declarations per module. `CommentText` may use subsections if substructure is needed.
- *Example:*

```
:- comment(appendix,"Other module functionality...").
```

- *The following properties should hold upon exit:*

  `CommentType` and `appendix` unify. (`= /2`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 9: :- `comment(CommentType,CommentText)`.

- *Description:* Provides a description of how the library should be loaded. Normally, this information is gathered automatically when generating documentation automatically. This declaration is meant for use when the module needs to be treated in some special way. There should be at most one of these declarations per module.
- *Example:*

```
:- comment(usage,"Do not use: still in development!").
```

- *The following properties should hold upon exit:*

  `CommentType` and `usage` unify. (`= /2`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 10: :- `comment(PredName,CommentText)`.

- *Description:* Provides an introductory comment for a given predicate, function, property, type, etc., denoted by `PredName`. When generating documentation for the module automatically, the text in `Text` will be used as the introduction of the corresponding predicate/function/... description. There should be at most one of these declarations per predicate, function, property, or type.
- *Example:*

```
:- comment(comment/2,"This declaration provides one of the main
   means for adding @concept{machine readable comments} to
   programs.").
```

- *The following properties should hold upon exit:*

  `PredName` is a Name/Arity structure denoting a predicate name:

```
prednáme(P/A) :-
        atm(P),
        nnegint(A).
```

  (`predname/1`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 11: :- `comment(CommentType,CommentText)`.

- *Description:* Documents a known *bug* or *planned improvement* in the module or application. Several of these declarations can appear per module. When generating documentation automatically, the text in the `Text` fields will be used as items in an itemized list of module or application bugs.
- *Example:*

```
:- comment(bug,"Comment text still has to be written by user.").
```

- *The following properties should hold upon exit:*

  `CommentType` and `bug` unify. (`= /2`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 12: :- `comment(Version,CommentText)`.

- *Description:* Provides a means for keeping a *log of changes.* `Version` contains the *version number* and date corresponding to the change and `CommentText` an explanation of the change. Several of these declarations can appear per module. When generating documentation automatically, the texts in the different `CommentText` fields typically appear as items in an itemized list of changes. The emacs Ciao mode helps tracking version numbers by prompting for version comments when files are saved. This mode requires version comments to appear in reverse chronological order (i.e., the topmost comment should be the most recent one).
- *Example:*

```
:- comment(version(1*1+21,1998/04/18,15:05*01+'EST'), "Added some
   missing comments.  (Manuel Hermenegildo)").
```

- *The following properties should hold upon exit:*

  `Version` is a complete version descriptor. (`version_descriptor/1`)

  `CommentText` is a *documentation string.* (`docstring/1`)

Usage 13: :- `comment(CommentType,VersionMaintenanceType)`.

- *Description:* Defines the type of version maintenance that should be performed by the `emacs` Ciao mode.
- *Example:*

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

Version control info is kept in directory `../version`. See the definition of `version_maintenance_type/1` for more information on the different version maintenance modes. See the documentation on the *emacs Ciao mode* in the Ciao manual for information on how to automatically insert version control `comment/2` declarations in files.

The version maintenance mode can also be set alternatively by inserting a comment such as:

```
%% Local Variables:
%% mode: CIAO
%% update-version-comments: "off"
%% End:
```

The lines above instruct emacs to put the buffer visiting the file in emacs Ciao mode and to turn version maintenance off. Setting the version maintenance mode in this way has the disadvantage that `lpdoc` will not be aware of the type of version maintenance being performed (the lines above are comments for Prolog). However, this can be useful in fact for setting the *version maintenance mode for packages* and other files meant for inclusion in other files, since that way the settings will not affect the file in which the package is included.

– *The following properties should hold upon exit:*

  `CommentType` and `version_maintenance` unify. (`= /2`)

  `VersionMaintenanceType` a type of version maintenance for a file. (`version_maintenance_type/1`)

Usage 14: :- `comment(CommentType,PredName)`.

– *Description:* This is a special case that is used to control which predicates are included in the documentation. Normally, only exported predicates are documented. A declaration `:- comment(doinclude,PredName).` forces documentation for predicate (or type, property, function, ...) `PredName` to be included even if `PredName` is not exported. Also, if `PredName` is reexported from another module, a declaration `:- comment(doinclude,PredName).` will force the documenation for `PredName` to appear directly in this module.

– *Example:*

```
:- comment(doinclude,p/3).
```

– *The following properties should hold upon exit:*

  `CommentType` and `doinclude` unify. (`= /2`)

  `PredName` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
```

  (`predname/1`)

Usage 15: :- `comment(CommentType,PredName)`.

– *Description:* A different usage which allows the second argument of `:- comment(doinclude,...)` to be a list of predicate names.

– *The following properties should hold upon exit:*

  `CommentType` and `doinclude` unify. (`= /2`)

  `PredName` is a list of `prednames`. (`list/2`)

Usage 16: :- `comment(CommentType,PredName)`.

- *Description:* This is similar to the previous usage but has the opposite effect: it signals that an exported predicate should *not* be included in the documentation.
- *Example:*

  ```
  :- comment(hide,p/3).
  ```
- *The following properties should hold upon exit:*

  `CommentType` and `hide` unify. (`= /2`)

  `PredName` is a Name/Arity structure denoting a predicate name:

  ```
  predname(P/A) :-
          atm(P),
          nnegint(A).
  ```
  (`predname/1`)

Usage 17: :- `comment(CommentType,PredName).`
- *Description:* A different usage which allows the second argument of `:- comment(hide,...)` to be a list of predicate names.
- *The following properties should hold upon exit:*

  `CommentType` and `hide` unify. (`= /2`)

  `PredName` is a list of `prednames`. (`list/2`)

Usage 18: :- `comment(CommentType,FileType).`
- *Description:* Provides a way of defining the intended use of the file. This use is normally easily inferred from the contents of the file itself, and therefore such a declaration is in general not needed. The exception is the special case of include files and Ciao packages, which are typically indistiguishable from normal *user* files (i.e., files which are not modules), but are however 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.). Typically, it is assumed by default that files which are not modules will be used as include files or packages. Thus, a `comment/2` declaration of this kind strictly only needs to be added to user-type files.

  Example:

  ```
  :- comment(filetype,user).
  ```

  There is another special case: the value `part`. This *filetype* is used to flag files which serve as introductions to boundaries between major *parts in large documents*. See Section 2.8.5 [Splitting large documents into parts], page 20 for details.
- *The following properties should hold upon exit:*

  `CommentType` and `filetype` unify. (`= /2`)

  `FileType` describes the intended use of a file. (`filetype/1`)

Usage 19: :- `comment(CommentType,FileName).`
- *Description:* Do not document anything that comes from a file whose name (after taking away the path and the suffix) is `FileName`. This is used for example when documenting packages to avoid the documenter from including documentation of certain other packages which the package being documented uses.
- *Example:*

  ```
  :- comment(nodoc,assertions).
  ```
- *The following properties should hold upon exit:*

  `CommentType` and `nodoc` unify. (`= /2`)

  `FileName` is an atom. (`atm/1`)

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

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.

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.

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.

4 The Ciao assertion package

Author(s): Manuel Hermenegildo, Francisco Bueno, German Puebla.

Version: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.5#8 (1999/12/9, 21:1:11 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 (assertions)

- **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].
- **New declarations defined:**

  `pred/1`, `pred/2`, `calls/1`, `calls/2`, `success/1`, `success/2`, `comp/1`, `comp/2`, `prop/1`, `prop/2`, `entry/1`, `modedef/1`, `decl/1`, `decl/2`, `comment/2`.
- **Other modules used:**
  - *System library modules:*

    `assertions/assertions_props`.
  - *Internal (engine) modules:*

    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `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 (assertions)

pred/1: DECLARATION

This assertion provides information on a predicate. The body of the assertion (its only argument) 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). All of them together cover all possible modes of usage.

For example, the following assertions describe (all the and the only) modes of usage of predicate `length/2` (see `lists`):

```
:- pred length(L,N) : list * var => list * integer
# "Computes the length of L.".
:- pred length(L,N) : var * integer => list * integer
# "Outputs L of length N.".
:- pred length(L,N) : list * integer => list * integer
# "Checks that L is of length N.".
```

Usage: :- `pred(AssertionBody).`

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

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

pred/2: DECLARATION

This assertion is similar to a `pred/1` assertion but it is explicitely qualified. Non-qualified `pred/1` assertions are assumed the qualifier `check`.

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

- *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

This assertion is similar to a `pred/1` assertion but it only provides information about the calls to a predicate. If one or several calls assertions are given they are understood to describe all possible calls to the predicate.

For example, the following assertion describes all possible calls to predicate `is/2` (see `arithmetic`):

```
:- calls is(term,arithexpression).
```

Usage: :- `calls(AssertionBody).`

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

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

calls/2: DECLARATION

This assertion is similar to a `calls/1` assertion but it is explicitely qualified. Non-qualified `calls/1` assertions are assumed the qualifier `check`.

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

- *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

This assertion is similar to a `pred/1` assertion but it only provides information about the answers to a predicate. The described answers might be conditioned to a particular way of calling the predicate.

For example, the following assertion specifies the answers of the `length/2` predicate *if* it is called as in the first mode of usage above (note that the previous pred assertion already conveys such information, however it also compelled the predicate calls, while the success assertion does not):

```
    :- success length(L,N) : list * var => list * integer.
```

Usage: :- `success(AssertionBody)`.

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

`AssertionBody` is a predicate assertion body. (`s_assrt_body/1`)

success/2: DECLARATION

This assertion is similar to a `success/1` assertion but it is explicitely qualified. Non-qualified `success/1` assertions are assumed the qualifier `check`.

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

– *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

This assertion is similar to a `pred/1` assertion but it only provides information about the global execution properties of a predicate (note that such kind of information is also conveyed by pred assertions). The described properties might be conditioned to a particular way of calling the predicate.

For example, the following assertion specifies that the computation of `append/3` (see `lists`) will not fail *if* it is called as described (but does not compel the predicate to be called that way):

```
    :- comp append(Xs,Ys,Zs) : var * var * var + not_fail.
```

Usage: :- `comp(AssertionBody)`.

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

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

comp/2: DECLARATION

This assertion is similar to a `comp/1` assertion but it is explicitely qualified. Non-qualified `comp/1` assertions are assumed the qualifier `check`.

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

– *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

This assertion is similar to a `pred/1` assertion but it flags that the predicate being documented is also a “ property.”

Properties are standard predicates, but which are *guaranteed to terminate for any possible instantiation state of their argument(s)*, do not perform side-effects which may interfere with the program behaviour, and do not further instantiate their arguments or add new constraints.

Provided the above holds, properties can thus be safely used as run-time checks. The program transformation used in `ciaopp` for run-time checking guarantees the third requirement. It also performs some basic checks on properties which in most cases are enough for the second requirement. However, it is the user's responsibility to guarantee termination of the properties defined. (See also Chapter 7 [Declaring regular types], page 57 for some considerations applicable to writing properties.)

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.

Usage: :- `prop(AssertionBody).`

– *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assrt_body/1`)

prop/2: DECLARATION

This assertion is similar to a `prop/1` assertion but it is explicitely qualified. Non-qualified `prop/1` assertions are assumed the qualifier `check`.

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

– *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`)

entry/1: DECLARATION

This assertion provides information about the *external* calls to a predicate. It is identical syntactically to a `calls/1` assertion. However, they describe only external calls, i.e., calls to the exported predicates of a module from outside the module, or calls to the predicates in a non-modular file from other files (or the user).

These assertions are *trusted* by the compiler. As a result, if their descriptions 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.

Usage: :- `entry(AssertionBody).`

– *The following properties should hold at call time:*
  `AssertionBody` is a call assertion body. (`c_assrt_body/1`)

modedef/1: DECLARATION

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. For 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.".
```

Usage: :- `modedef(AssertionBody)`.

– *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assrt_body/1`)

decl/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it is used for declarations instead than for predicates.

Usage: :- `decl(AssertionBody)`.

– *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assrt_body/1`)

decl/2: DECLARATION

This assertion is similar to a `decl/1` assertion but it is explicitely qualified. Non-qualified `decl/1` assertions are assumed the qualifier `check`.

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

– *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 (`assertions`)

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. See also ⟨undefined⟩ [Run-time checking of assertions], page ⟨undefined⟩.

- *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: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.7#156 (2001/11/24, 13:23:30 CET)

This module is part of the `assertions` library. It provides the formal definition of the syntax of several forms of assertions and describes their meaning. It does so by defining types and properties related to the assertions themselves. The text describes, for example, the overall fields which are admissible in the bodies of assertions, where properties can be used inside these bodies, how to combine properties for a given predicate argument (e.g., conjunctions) , etc. and provides some examples.

5.1 Usage and interface (`assertions_props`)

- **Library usage:**
  `:- use_module(library(assertions_props)).`
- **Exports:**
  - *Properties:*
    `head_pattern/1`, `nabody/1`, `docstring/1`.
  - *Regular Types:*
    `assrt_body/1`, `complex_arg_property/1`, `property_conjunction/1`, `property_starterm/1`, `complex_goal_property/1`, `dictionary/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`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

5.2 Documentation on exports (`assertions_props`)

assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `decl/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.

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

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.

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.

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 implicitely 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.

nabody/1: PROPERTY

Usage: `nabody(ABody)`

– *Description:* `ABody` is a normalized assertion body.

dictionary/1: REGTYPE

Usage: `dictionary(D)`

– *Description:* `D` is a dictionary of variable names.

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.

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.

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.

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.

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.

predfunctor/1: REGTYPE

Usage: `predfunctor(X)`

– *Description:* `X` is a type of assertion which defines a predicate.

propfunctor/1: REGTYPE

Usage: `propfunctor(X)`

– *Description:* `X` is a type of assertion which defines a *property*.

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 Basic data types and properties

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.7#204 (2002/4/22, 18:42:18 CEST)

This library contains the set of basic properties used by the builtin predicates, and which constitute the basic data types and properties of the language. They can be used both as type testing builtins within programs (by calling them explicitly) and as properties in assertions.

6.1 Usage and interface (basic_props)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Properties:*
    member/2, compat/2, iso/1, not_further_inst/2, regtype/1.
  - *Regular Types:*
    term/1, int/1, nnegint/1, flt/1, num/1, atm/1, struct/1, gnd/1, constant/1, callable/1, operator_specifier/1, list/1, list/2, sequence/2, sequence_or_list/2, character_code/1, string/1, predname/1, atm_or_atm_list/1.
- **Other modules used:**
  - *Internal (engine) modules:*
    arithmetic, atomic_basic, attributes, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_basic, term_compare, term_typing.

6.2 Documentation on exports (basic_props)

term/1: REGTYPE

The most general type (includes all possible terms).

Usage 1: `term(X)`

- *Description:* `X` is any term.

Usage 2: `term(X)`

- *Description:* `X` is any term.

int/1: REGTYPE

The type of integers. The range of integers is `[-2^2147483616, 2^2147483616)`. Thus for all practical purposes, the range of integers can be considered infinite.

Usage 1: `int(T)`

- *Description:* `T` is an integer.

Usage 2: `int(T)`

- *Description:* `T` is an integer.

nnegint/1: REGTYPE

The type of non-negative integers, i.e., natural numbers.

Usage 1: `nnegint(T)`

– *Description:* `T` is a non-negative integer.

Usage 2: `nnegint(T)`

– *Description:* `T` is a non-negative integer.

flt/1: REGTYPE

The type of floating-point numbers. The range of floats is the one provided by the C `double` type, typically `[4.9e-324, 1.8e+308]` (plus or minus). There are also three special values: Infinity, either positive or negative, represented as `1.0e1000` and `-1.0e1000`; and Not-a-number, which arises as the result of indeterminate operations, represented as `0.Nan`

Usage 1: `flt(T)`

– *Description:* `T` is a float.

Usage 2: `flt(T)`

– *Description:* `T` is a float.

num/1: REGTYPE

The type of numbers, that is, integer or floating-point.

Usage 1: `num(T)`

– *Description:* `T` is a number.

Usage 2: `num(T)`

– *Description:* `T` is a number.

atm/1: REGTYPE

The type of atoms, or non-numeric constants. The size of atoms is unbound.

Usage 1: `atm(T)`

– *Description:* `T` is an atom.

Usage 2: `atm(T)`

– *Description:* `T` is an atom.

struct/1: REGTYPE

The type of compound terms, or terms with non-zeroary functors. By now there is a limit of 255 arguments.

Usage 1: `struct(T)`

– *Description:* `T` is a compound term.

Usage 2: `struct(T)`

– *Description:* `T` is a compound term.

gnd/1: REGTYPE

The type of all terms without variables.

Usage 1: `gnd(T)`

– *Description:* `T` is ground.

Usage 2: `gnd(T)`

– *Description:* `T` is ground.

constant/1: REGTYPE

Usage 1: `constant(T)`

– *Description:* `T` is an atomic term (an atom or a number).

Usage 2: `constant(T)`

– *Description:* `T` is an atomic term (an atom or a number).

callable/1: REGTYPE

Usage 1: `callable(T)`

– *Description:* `T` is a term which represents a goal, i.e., an atom or a structure.

Usage 2: `callable(T)`

– *Description:* `T` is a term which represents a goal, i.e., an atom or a structure.

operator_specifier/1: REGTYPE

The type and associativity of an operator is described by the following mnemonic atoms:

`xfx`	Infix, non-associative: it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself.
`xfy`	Infix, right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator.
`yfx`	Infix, left-associative: same as above, but the other way around.
`fx`	Prefix, non-associative: the subexpression must be of *lower* precedence than the operator.
`fy`	Prefix, associative: the subexpression can be of the *same* precedence as the operator.
`xf`	Postfix, non-associative: the subexpression must be of *lower* precedence than the operator.
`yf`	Postfix, associative: the subexpression can be of the *same* precedence as the operator.

Usage 1: `operator_specifier(X)`

– *Description:* `X` specifies the type and associativity of an operator.

Usage 2: `operator_specifier(X)`

– *Description:* `X` specifies the type and associativity of an operator.

list/1: REGTYPE

A list is formed with successive applications of the functor `'.'/2`, and its end is the atom `[]`. Defined as

```
list([]).
list([_1|L]) :-
        list(L).
list([]).
list([_1|L]) :-
        list(L).
```

Usage 1: `list(L)`

– *Description:* `L` is a list.

Usage 2: `list(L)`

– *Description:* `L` is a list.

list/2: REGTYPE

`list(L,T)`

`L` is a list, and for all its elements, `T` holds.

Meta-predicate with arguments: `list(?,pred(1))`.

Usage 1: `list(L,T)`

– *Description:* `L` is a list of `T`s.

Usage 2: `list(L,T)`

– *Description:* `L` is a list of `T`s.

member/2: PROPERTY

Usage 1: `member(X,L)`

– *Description:* `X` is an element of `L`.

Usage 2: `member(X,L)`

– *Description:* `X` is an element of `L`.

sequence/2: REGTYPE

A sequence is formed with zero, one or more occurrences of the operator `','/2`. For example, `a, b, c` is a sequence of three atoms, `a` is a sequence of one atom.

Meta-predicate with arguments: `sequence(?,pred(1))`.

Usage 1: `sequence(S,T)`

– *Description:* `S` is a sequence of `T`s.

Usage 2: `sequence(S,T)`

– *Description:* `S` is a sequence of `T`s.

sequence_or_list/2: REGTYPE

Meta-predicate with arguments: `sequence_or_list(?,pred(1))`.

Usage 1: `sequence_or_list(S,T)`

– *Description:* `S` is a sequence or list of `T`s.

Usage 2: `sequence_or_list(S,T)`

– *Description:* `S` is a sequence or list of `T`s.

character_code/1: REGTYPE

Usage 1: `character_code(T)`

– *Description:* `T` is an integer which is a character code.

– *The following properties hold upon exit:*

`T` is an integer. (`int/1`)

Usage 2: `character_code(T)`

– *Description:* `T` is an integer which is a character code.

– *The following properties hold upon exit:*

`T` is an integer. (`int/1`)

string/1: REGTYPE

A string is a list of character codes. The usual syntax for strings `"string"` is allowed, which is equivalent to `[0's,0't,0'r,0'i,0'n,0'g]` or `[115,116,114,105,110,103]`. There is also a special Ciao syntax when the list is not complete: `"st"||R` is equivalent to `[0's,0't|R]`.

Usage 1: `string(T)`

– *Description:* `T` is a string (a list of character codes).

– *The following properties hold upon exit:*

`T` is a list of `character_code`s. (`list/2`)

Usage 2: `string(T)`

– *Description:* `T` is a string (a list of character codes).

– *The following properties hold upon exit:*

`T` is a list of `character_code`s. (`list/2`)

predname/1: REGTYPE

Usage 1: `predname(P)`

– *Description:* `P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
predname(P/A) :-
        atm(P),
        nnegint(A).
```

Usage 2: `predname(P)`

– *Description:* `P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
predname(P/A) :-
        atm(P),
        nnegint(A).
```

atm_or_atm_list/1: REGTYPE

Usage 1: `atm_or_atm_list(T)`

– *Description:* `T` is an atom or a list of atoms.

Usage 2: `atm_or_atm_list(T)`

– *Description:* `T` is an atom or a list of atoms.

compat/2: PROPERTY

This property captures the notion of type or property compatibility. The instantiation or constraint state of the term is compatible with the given property, in the sense that assuming that imposing that property on the term does not render the store inconsistent. For example, terms `X` (i.e., a free variable), `[Y|Z]`, and `[Y,Z]` are all compatible with the regular type `list/1`, whereas the terms `f(a)` and `[1|2]` are not.

Meta-predicate with arguments: `compat(?,pred(1))`.

Usage 1: `compat(Term,Prop)`

– *Description:* `Term` is *compatible* with `Prop`

Usage 2: `compat(Term,Prop)`

– *Description:* `Term` is *compatible* with `Prop`

iso/1: PROPERTY

Usage 1: `iso(G)`

– *Description: Complies with the ISO-Prolog standard.*

Usage 2: `iso(G)`

– *Description: Complies with the ISO-Prolog standard.*

not_further_inst/2: PROPERTY

Usage 1: `not_further_inst(G,V)`

– *Description:* `V` is not further instantiated.

Usage 2: `not_further_inst(G,V)`

– *Description:* `V` is not further instantiated.

regtype/1: PROPERTY

Usage 1: `regtype(G)`

– *Description:* Defines a regular type.

Usage 2: `regtype(G)`

– *Description:* Defines a regular type.

7 Declaring regular types

Author(s): Manuel Hermenegildo, Pedro Lopez, Francisco Bueno.

Version: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.5#9 (1999/12/9, 21:57:42 MET)

This library package adds some new declaration definitions and new operator definitions to user programs. These new declarations and operators provide some very simple syntactic sugar to support regular type definitions in source code. Regular types are just properties which have the additional characteristic of being regular types (`basic_props:regtype/1`).

For example, this library package allows writing:

```
:- regtype tree(X) # "X is a tree.".
```

instead of the more combersome:

```
:- prop tree(X) + regtype # "X is a tree.".
```

Regular types can be used as properties to describe predicates and play an essential role in program debugging (see the Ciao Prolog preprocessor (`ciaopp`) manual).

In this chapter we explain some general considerations worth taking into account when writing properties in general, not just regular types. The exact syntax of regular types is also described.

7.1 Defining properties

Given the classes of assertions in the Ciao assertion language, there are two fundamental classes of properties. Properties used in assertions which refer to execution states (i.e., `calls/1`, `success/1`, and the like) are called *properties of execution states*. Properties used in assertions related to computations (i.e., `comp/1`) are called *properties of computations*. Different considerations apply when writing a property of the former or of the later kind.

Consider a definition of the predicate `string_concat/3` which concatenates two character strings (represented as lists of ASCII codes):

```
string_concat([],L,L).
string_concat([X|Xs],L,[X|NL]):- string_concat(Xs,L,NL).
```

Assume that we would like to state in an assertion that each argument "is a list of integers." However, we must decide which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list of integers" (let us call this property `instantiated_to_intlist/1`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list of integers" (we will call this property `compatible_with_intlist/1`). For example, `instantiated_to_intlist/1` should be true for the terms `[]` and `[1,2]`, but should not for `X`, `[a,2]`, and `[X,2]`. In turn, `compatible_with_intlist/1` should be true for `[]`, `X`, `[1,2]`, and `[X,2]`, but should not be for `[X|1]`, `[a,2]`, and `1`. We refer to properties such as `instantiated_to_intlist/1` above as *instantiation properties* and to those such as `compatible_with_intlist/1` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types").

It turns out that both of these notions are quite useful in practice. In the example above, we probably would like to use `compatible_with_intlist/1` to state that on success of `string_concat/3` all three argument must be compatible with lists of integers in an assertion like:

```
:- success string_concat(A,B,C) => ( compatible_with_intlist(A),
                                     compatible_with_intlist(B),
                                     compatible_with_intlist(C) ).
```

With this assertion, no error will be flagged for a call to `string_concat/3` such as `string_concat([20],L,R)`, which on success produces the resulting atom `string_concat([20],L,[20|L])`, but a call `string_concat([],a,R)` would indeed flag an error.

On the other hand, and assuming that we are running on a Prolog system, we would probably like to use `instantiated_to_intlist/1` for `sumlist/2` as follows:

```
:- calls sumlist(L,N) : instantiated_to_intlist(L).

sumlist([],0).
sumlist([X|R],S) :- sumlist(R,PS), S is PS+X.
```

to describe the type of calls for which the program has been designed, i.e., those in which the first argument of `sumlist/2` is indeed a list of integers.

The property `instantiated_to_intlist/1` might be written as in the following (Prolog) definition:

```
:- prop instantiated_to_intlist/1.

instantiated_to_intlist(X) :-
       nonvar(X), instantiated_to_intlist_aux(X).

instantiated_to_intlist_aux([]).
instantiated_to_intlist_aux([X|T]) :-
       integer(X), instantiated_to_intlist(T).
```

(Recall that the Prolog builtin `integer/1` itself implements an instantiation check, failing if called with a variable as the argument.)

The property `compatible_with_intlist/1` might in turn be written as follows (also in Prolog):

```
:- prop compatible_with_intlist/1.

compatible_with_intlist(X) :- var(X).
compatible_with_intlist(X) :-
       nonvar(X), compatible_with_intlist_aux(X).

compatible_with_intlist_aux([]).
compatible_with_intlist_aux([X|T]) :-
       int_compat(X), compatible_with_intlist(T).

int_compat(X) :- var(X).
int_compat(X) :- nonvar(X), integer(X).
```

Note that these predicates meet the criteria for being properties and thus the `prop/1` declaration is correct.

Ensuring that a property meets the criteria for "not affecting the computation" can sometimes make its coding somewhat tedious. In some ways, one would like to be able to write simply:

```
intlist([]).
intlist([X|R]) :- int(X), intlist(R).
```

(Incidentally, note that the above definition, provided that it suits the requirements for being a property and that `int/1` is a regular type, meets the criteria for being a regular type. Thus, it could be declared `:- regtype intlist/1`.)

But note that (independently of the definition of `int/1`) the definition above is not the correct instantiation check, since it would succeed for a call such as `intlist(X)`. In fact, it is not strictly correct as a compatibility property either, because, while it would fail or succeed as expected, it would perform instantiations (e.g., if called with `intlist(X)` it would bind `X` to `[]`). In practice, it is convenient to provide some run-time support to aid in this task.

The run-time support of the Ciao system (see ⟨undefined⟩ [Run-time checking of assertions], page ⟨undefined⟩) ensures that the execution of properties is performed in such a way that properties written as above can be used directly as instantiation checks. Thus, writing:

```
:- calls sumlist(L,N) : intlist(L).
```

has the desired effect. Also, the same properties can often be used as compatibility checks by writing them in the assertions as `compat(Property)` (`basic_props:compat/1`). Thus, writing:

```
:- success string_concat(A,B,C) => ( compat(intlist(A)),
                                     compat(intlist(B)),
                                     compat(intlist(C)) ).
```

also has the desired effect.

As a general rule, the properties that can be used directly for checking for compatibility should be *downwards closed*, i.e., once they hold they will keep on holding in every state accessible in forwards execution. There are certain predicates which are inherently *instantiation* checks and should not be used as *compatibility* properties nor appear in the definition of a property that is to be used with `compat`. Examples of such predicates (for Prolog) are `==`, `ground`, `nonvar`, `integer`, `atom`, `>`, etc. as they require a certain instantiation degree of their arguments in order to succeed.

In contrast with properties of execution states, *properties of computations* refer to the entire execution of the call(s) that the assertion relates to. One such property is, for example, `not_fail/1` (note that although it has been used as in `:- comp append(Xs,Ys,Zs) + not_fail`, it is in fact read as `not_fail(append(Xs,Ys,Zs))`; see `assertions_props:complex_goal_property/1`). For this property, which should be interpreted as "execution of the predicate either succeeds at least once or loops," we can use the following predicate `not_fail/1` for run-time checking:

```
not_fail(Goal):-
      if( call(Goal),
          true,            %% then
          warning(Goal) ). %% else
```

where the `warning/1` (library) predicate simply prints a warning message.

In this simple case, implementation of the predicate is not very difficult using the (non-standard) `if/3` builtin predicate present in many Prolog systems.

However, it is not so easy to code predicates which check other properties of the computation and we may in general need to program a meta-interpreter for this purpose.

7.2 Usage and interface (`regtypes`)

- **Library usage:**
  `:- use_package(regtypes).`
  or
  `:- module(...,...,[regtypes]).`
- **New operators defined:**
  `regtype/1` [1150,fx], `regtype/2` [1150,xfx].
- **New declarations defined:**
  `regtype/1`, `regtype/2`.
- **Other modules used:**
  - *System library modules:*
    `assertions/assertions_props`.
  - *Internal (engine) modules:*
    `basiccontrol`.

7.3 Documentation on new declarations (`regtypes`)

regtype/1: DECLARATION

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, ..., v_n)  :- body_1, ..., 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` >= 0 and `p/n` is a *parametric type functor* (whereas the predicate defined by the clauses is `p/n+1`).
4. `v_1`, ..., `v_n` are unique variables, which are called *parametric variables.*
5. Each `body_i` is of the form:
   1. `t(z)` where `z` is one of the *term variables* and `t` is a *regular type expression*;
   2. `q(y, t_1, ..., t_m)` where `m` >= 0, `q/m` is a *parametric type functor*, not in the set of functors `=/2`, `^/2`, `./3`.
      `t_1, ..., 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 (but regular type abstractions might also be used in some cases, see Chapter 9 [Meta-properties], page 67).

A parametric type functor is a regular type, defined by a regular program, or a basic type. Basic types are defined in Chapter 6 [Basic data types and properties], page 51.

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.

Usage: :- `regtype(AssertionBody)`.

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

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

regtype/2: DECLARATION

This assertion is similar to a `regtype/1` assertion but it is explicitely qualified. Non-qualified `regtype/1` assertions are assumed the qualifier `check`. Note that checking regular type definitions should be done with the `ciaopp` preprocessor.

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

- *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`)

8 Properties which are native to analyzers

Author(s): Francisco Bueno, Manuel Hermenegildo, Pedro Lopez.

Version: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.5#1 (1999/11/29, 17:12:34 MET)

This library contains a set of properties which are natively understood by the different program analyzers of `ciaopp`. They are used by `ciaopp` on output and they can also be used as properties in assertions.

8.1 Usage and interface (`native_props`)

- **Library usage:**
  `:- use_module(library('assertions/native_props'))`
  or also as a package `:- use_package(nativeprops)`.
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `linear/1`, `mshare/1`, `fails/1`, `not_fails/1`, `possibly_fails/1`, `covered/1`, `not_covered/1`, `is_det/1`, `possibly_nondet/1`, `mut_exclusive/1`, `not_mut_exclusive/1`, `size_lb/2`, `size_ub/2`, `steps_lb/2`, `steps_ub/2`, `sideff_pure/1`, `sideff_soft/1`, `sideff_hard/1`.
- **Other modules used:**
  - *System library modules:*
    `andprolog/andprolog_rt`, `terms_vars`, `sort`, `lists`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

8.2 Documentation on exports (`native_props`)

linear/1: PROPERTY

`linear(X)`

`X` is bound to a term which is linear, i.e., if it contains any variables, such variables appear only once in the term. For example, `[1,2,3]` and `f(A,B)` are linear terms, while `f(A,A)` is not.

Usage: `linear(X)`

- *Description:* `X` is instantiated to a linear term.

mshare/1: PROPERTY

`mshare(X)`

`X` contains all *sharing sets* [JL88,MH89] which specify the possible variable occurrences in the terms to which the variables involved in the clause may be bound. Sharing sets are a compact way of representing groundness of variables and dependencies between variables. This representation is however generally difficult to read for humans. For this reason, this information is often translated to `ground/1`, `indep/1` and `indep/2` properties, which are easier to read.

Usage: `mshare(X)`

– *Description:* The sharing pattern is `X`.

fails/1: PROPERTY

`fails(X)`

Calls of the form `X` fail.

Usage: `fails(X)`

– *Description:* Calls of the form `X` fail.

not_fails/1: PROPERTY

`not_fails(X)`

Calls of the form `X` produce at least one solution, or not terminate [DLGH97].

Usage: `not_fails(X)`

– *Description:* All the calls of the form `X` do not fail.

possibly_fails/1: PROPERTY

`possibly_fails(X)`

Non-failure is not ensured for any call of the form `X` [DLGH97]. In other words, nothing can be ensured about non-failure nor termination of such calls.

Usage: `possibly_fails(X)`

– *Description:* Non-failure is not ensured for calls of the form `X`.

covered/1: PROPERTY

`covered(X)`

For any call of the form `X` there is at least one clause whose test succeeds (i.e. all the calls of the form `X` are covered.) [DLGH97].

Usage: `covered(X)`

– *Description:* All the calls of the form `X` are covered.

not_covered/1: PROPERTY

`not_covered(X)`

There is some call of the form `X` for which there is not any clause whose test succeeds [DLGH97].

Usage: `not_covered(X)`

– *Description:* Not all of the calls of the form `X` are covered.

is_det/1: PROPERTY

`is_det(X)`

All calls of the form `X` are deterministic, i.e. produce at most one solution, or not terminate.

Usage: `is_det(X)`

– *Description:* All calls of the form `X` are deterministic.

possibly_nondet/1: PROPERTY

`possibly_nondet(X)`

Non-determinism is not ensured for all calls of the form `X`. In other words, nothing can be ensured about determinacy nor termination of such calls.

Usage: `possibly_nondet(X)`

– *Description:* Non-determinism is not ensured for calls of the form `X`.

mut_exclusive/1: PROPERTY

`mut_exclusive(X)`

For any call of the form `X` at most one clause succeeds, i.e. clauses are pairwise exclusive.

Usage: `mut_exclusive(X)`

– *Description:* For any call of the form `X` at most one clause succeeds.

not_mut_exclusive/1: PROPERTY

`not_mut_exclusive(X)`

Not for all calls of the form `X` at most one clause succeeds. I.e. clauses are not disjoint for some call.

Usage: `not_mut_exclusive(X)`

– *Description:* Not for all calls of the form `X` at most one clause succeeds.

size_lb/2: PROPERTY

`size_lb(X,Y)`

The minimum size of the terms to which the argument `Y` is bound to is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93].

Usage: `size_lb(X,Y)`

– *Description:* `Y` is a lower bound on the size of argument `X`.

size_ub/2: PROPERTY

`size_ub(X,Y)`

The maximum size of the terms to which the argument `Y` is bound to is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93].

Usage: `size_ub(X,Y)`

– *Description:* `Y` is a upper bound on the size of argument `X`.

steps_lb/2: PROPERTY

`steps_lb(X,Y)`

The minimum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DLGHL97,LGHD96]

Usage: `steps_lb(X,Y)`

– *Description:* `Y` is a lower bound on the cost of any call of the form `X`.

steps_ub/2: PROPERTY

`steps_ub(X,Y)`

The maximum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DL93,LGHD96]

Usage: `steps_ub(X,Y)`

– *Description:* `Y` is a upper bound on the cost of any call of the form `X`.

sideff_pure/1: PROPERTY

Usage: `sideff_pure(X)`

– *Description:* `X` is pure, i.e., has no side-effects.

sideff_soft/1: PROPERTY

Usage: `sideff_soft(X)`

– *Description:* `X` has *soft side-effects*, i.e., those not affecting program execution (e.g., input/output).

sideff_hard/1: PROPERTY

Usage: `sideff_hard(X)`

– *Description:* `X` has *hard side-effects*, i.e., those that might affect program execution (e.g., assert/retract).

indep/1: PREDICATE

Usage: `indep(X)`

– *Description:* The variables in pairs in `X` are pairwise independent.

indep/2: PROPERTY

Usage: `indep(X,Y)`

– *Description:* `X` and `Y` do not have variables in common.

9 Meta-properties

Author(s): Francisco Bueno.

Version: 1.7#208 (2002/4/23, 19:9:14 CEST)

Version of last change: 1.7#167 (2002/1/3, 17:43:50 CET)

This library allows the use of some meta-constructs which provide for specifying properties of terms which are unknown at the time of the specification, or expressed with a shorthand for the property definition, i.e., without really defining it.

An example of such use is an assertion which specifies that any property holding upon call will also hold upon exit:

```
:- pred p(X) : Prop(X) => Prop(X).
```

Another example is using shorthands for properties when documenting:

```
:- pred p(X) : regtype(X,(^(list;list);list)).
```

(See below for an explanation of such a regular type.)

9.1 Usage and interface (`meta_props`)

- **Library usage:**
  `:- use_module(library('assertions/meta_props'))`
  or also as a package `:- use_package(metaprops)`.
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `call/2`, `prop/2`, `regtype/2`.
  - *Multifiles:*
    `callme/2`.
- **Other modules used:**
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

9.2 Documentation on exports (`meta_props`)

call/2: PROPERTY

`call(P,A)`

`A` has property `P` (provided that `P` is a property). Equivalent to `P(A)`.

Usage: `call(P,A)`

- *Description:* `A` has property `P`.
- *If the following properties hold at call time:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)

prop/2: PROPERTY

Usage: `prop(A,P)`

- *Description:* `A` has property `P`.
- *If the following properties hold at call time:*

  `P` has property `^ (callable;prop_abs)`. (`prop/2`)

regtype/2: PROPERTY

Usage: `regtype(A,T)`

- *Description:* `A` is of type `T`.
- *If the following properties hold at call time:*

  `T` has property `^ (regtype;prop_abs)`. (`prop/2`)

9.3 Documentation on multifiles (`meta_props`)

callme/2: PREDICATE

(User defined.) A hook predicate you have to define as `callme(P,X):- P(X), !.` in the program that uses this library. This is done automatically if the package is used instead of the library module (but then you *should not* define `callme/2` in your program).

The predicate is *multifile*.

9.4 Documentation on internals (`meta_props`)

prop_abs/1: PROPERTY

`prop_abs(Prop)`

`Prop` is a *property abstraction*, i.e., a *parametric property*, or a term formed of property abstractions, where the functors used in the term are escaped by `^`.

One particular case of property abstractions are *parametric regular type abstractions*, i.e., a parametric type functor or a `^`-escaped term formed of regular type abstractions.

Such abstractions are a short-hand for a corresponding regular type (correspondingly, property). For example, the following abstraction:

```
^(list;list);list
```

denotes terms of the form `(X;Y)` where `list(X)` and `list(Y)` hold and also terms `T` such that `list(T)` holds. It is equivalent to the regular type:

```
abstract_type((X;Y)):- list(X), list(Y).
abstract_type(T):- list(T).
```

Usage: `prop_abs(Prop)`

- *Description:* `Prop` is a property abstraction.

10 An Example - Documenting a Library Module

Author(s): Manuel Hermenegildo.

Version: 1.9#58 (2002/4/19, 20:59:33 CEST)

Version of last change: 1.9#39 (1999/12/9, 21:2:34 MET)

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,hiord]).

%% Some properties that the current CiaoPP analyzers understand natively:
:- use_module(library('assertions/native_props')).

%% 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).

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

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

%% :- doc can be used in place of :- comment.
:- doc(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(~call(T),~tree_of(T),~tree_of(T)).

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

:- 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,not_fails)
        # "This mode is nice.".
:- pred p(Preds,Value,Assoc)
        : var * var * list
       => int * int * list
        + not_fails # "This mode is also nice.".
:- pred p/3
       => list * int * list
        + (not_fails,not_fails)
        # "Just playing around.".

:- pred q(A)
        : list(A)
       => (list(A),gnd(A))
        + not_fails
        # "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))
        + not_fails
        # "This uses parametric types".

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

:- 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)
        :: T(A)
        => 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))
        +  not_fails
        #  "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) + steps_lb(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:
```

11 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.*

11.1 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:*
    `assertions/native_props`, `engine/basic_props`.
  - *Internal (engine) modules:*
    `hiord_rt`, `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

11.2 Documentation on exports (`example_module`)

bar/1: REGTYPE

Usage: `bar(X)`

- *Description:* `X` is an acceptable kind of bar.

baz/1: REGTYPE

A regular type, defined as follows:

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

aorb/1: REGTYPE

A regular type, defined as follows:

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

tree_of/2: REGTYPE

A regular type, defined as follows:

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

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).
```

list/1: REGTYPE

Usage 1: `list(L)`

– *Description:* `L` is a list.

Usage 2: `list(L)`

– *Description:* `L` is a list.

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:*
  All the calls of the form `q(A)` do not fail. (`not_fails/1`)

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 *data*.

Usage: `r(A)`

- *Description:* This uses parametric types
- *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 of `int`s. (`list/2`)

  `A` is ground. (`gnd/1`)
- *The following properties should hold globally:*

  All the calls of the form `r(A)` do not fail. (`not_fails/1`)

og/1: PREDICATE

No further documentation available for this predicate.

t/5: PREDICATE

Usage: `t(+A,-B,?(C),@(D),og(E))`

- *Description:* This predicate uses *modes* extensively.
- *Call and exit should be compatible with:*

  `+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`)

- *The following properties should hold at call time:*
  `B` is rather long. (`long/1`)
- *The following properties should hold upon exit:*
  `C` is ground. (`gnd/1`)
  `A` is ground. (`gnd/1`)
- *The following properties should hold globally:*
  All the calls of the form `t(+A,-B,?(C),@(D),og(E))` do not fail. (`not_fails/1`)

u/3: PREDICATE

Usage 2:

- *Call and exit should be compatible with:*
  `Arg1` is an integer. (`int/1`)
  `Arg2` is a list of `mytypes`. (`list/2`)
  `Arg3` is an integer. (`int/1`)

w/1: PREDICATE

p/5: PREDICATE

Usage: `p(og(int),in,@(list(int)),-,+A)`

- *The following properties should hold globally:*
  `1+length(A)` is a lower bound on the cost of any call of the form `p(og(int),in,@(list(int)),-,+A)`. (`steps_lb/2`)

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: REGTYPE

Usage 1: `list(L)`

- *Description:* `L` is a list.

Usage 2: `list(L)`

- *Description:* `L` is a list.

11.3 Documentation on multifiles (`example_module`)

p/3: PREDICATE

A **general comment** on the predicate.

The predicate is *multifile*.

The predicate is 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:
  All the calls of the form `p(Arg1,Arg2,Arg3)` do not fail. (`not_fails/1`)

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`)
  All the calls of the form `p(Arg1,Arg2,Arg3)` do not fail. (`not_fails/1`)

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:*
  All the calls of the form `p(Preds,Value,Assoc)` do not fail. (`not_fails/1`)

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:*
  All the calls of the form `p(Arg1,Arg2,Arg3)` do not fail. (`not_fails/1`)
  All the calls of the form `p(Arg1,Arg2,Arg3)` do not fail. (`not_fails/1`)

11.4 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:*
  All the calls of the form `s(A)` do not fail. (`not_fails/1`)

list/2: REGTYPE

Usage 1: `list(L,T)`

- *Description:* `L` is a list of `T`s.

Usage 2: `list(L,T)`

- *Description:* `L` is a list of `T`s.

og/2: MODE

Usage: `og(A,T)`

- *Description:* This is a *parametric mode definition.*
- *Call and exit are compatible with:*
  `call(T,A)` (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`)

12 Installing lpdoc

Author(s): Manuel Hermenegildo.

Version: 1.9#58 (2002/4/19, 20:59:33 CEST)

Version of last change: 1.9#40 (1999/12/9, 21:3:59 MET)

The source distribution contains all the source code and libraries and can be compiled on a supported Prolog system (`lpdoc` is developed using `Ciao` Prolog). The latest publically distributed version of `lpdoc` is available from `http::/www.clip.dia.fi.upm.es/Software/Ciao`. A newer version in Beta test is often available in `http::/www.clip.dia.fi.upm.es/Software/Beta/Ciao`.

12.1 Installing the source distribution

- Before installing `lpdoc`, you may want to read Section 12.2 [Other software packages required], page 83. Make sure that `emacs` is installed in your system
- Uncompress (using `gunzip`) and unpackage (using `tar -xpf`) the distribution in a suitable directory. This will create a new directory called `lpdoc` as well as a link `lpdoc-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 `lpdoc` (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 `lpdoc` binaries, libraries, and documents installed, by setting the corresponding variables in `SETTINGS`.
- Type `gmake install`. This should create the `lpdoc` executable and install it in the `BINDIR` directory, install the `lpdoc` library in a separate directory in the `LIBDIR` directory, and install the `lpdoc` documentation in the `DOCDIR` directory.
- In order for the `lpdoc` 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 `lpdoc` library.

12.2 Other software packages required

The most basic functionality of `lpdoc` (generating manuals in `.texi` format, short manua entries in `.manl` format, generating *index* files) is essentially self contained. However, using the full capabilities of `lpdoc` requires having several other software packages installed in the system. Fortunately, all of these packages are public domain software and they will normally be already installed in, e.g., a standard `Linux` distribution. It should be relatively easy to get and install the required packages in other Unix-like packages or even in Windows, under the Cygwin environment.

- **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. 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`. Generating the `.dvi` file requires that the `texinfo.tex` file (containing the relevant macros) be available to `tex`. This file is normally included with modern `tex` distributions, although it may be obsolete. An appropriate and up-to-date one for `lpdoc` is provided with the `lpdoc` distribution, stored in the `lpdoc` library during installation, and used automatically when `lpdoc` runs `tex`. 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.

PART II - LPdoc Internals Manual

13 Documentation generation library

Author(s): Manuel Hermenegildo.

Version: 1.9#58 (2002/4/19, 20:59:33 CEST)

Version of last change: 1.9#35 (1999/12/9, 0:29:7 CET)

This library provides some predicates which generate documentation automatically for a given module or application, using the declarations and assertions used in the module itself as input (see the `assertions` library). By default, only the exported predicates of the module appear in the documentation. The predicates will be documented in the order in which they appear in the `module/1` or `module/2` declaration.

The idea of this package is on one hand to reuse the information present in the assertions and on the other to help ensure that code and documentation are kept as coherent as possible. Hopefully, keeping them close together should help in this always difficult task. The resulting documentation is somewhat rigidly structured, but generally sufficient for a *reference* manual, provided a little effort is put into the assertions and comments. The end product understandably depends heavily on how much work is put into adding additional comments to the source. Some documentation will be generated in any case, but it is recommended that, at the minimum, a module title and a comment for each of the exported predicates be provided.

The exact format in which the documentation is generated is defined in an imported module `autodocformats`. See the description of the imported predicates for more details and descriptions of the interface. A default definition of this module is provided in the `autodocformats` library. A simple example of the use of this library for generating a `texinfo` reference manual (including a driver script, useful Makefiles, etc.) is included with the library source code. Other examples can be found in the Ciao documentation directory (i.e., the Ciao manuals themselves).

13.1 Usage and interface (autodoc)

- **Library usage:**
  `:- use_module(library(autodoc)).`
- **Exports:**
  - *Predicates:*
    `autodoc/10`, `generate_man_page/5`, `generate_description/6`, `generate_info_dir_entry/5`, `rewrite_docstring/4`.
  - *Regular Types:*
    `modtype/1`.
  - *Multifiles:*
    `library_directory/1`.
- **Other modules used:**
  - *Application modules:*
    `lpdoclib(autodocformats)`, `lpdoclib(comments)`.
  - *System library modules:*
    `streams`, `ttyout`, `format`, `aggregates`, `read`, `errhandle`, `dec10_io`, `compiler/compiler`, `assertions/assrt_lib`, `compiler/c_itf`, `assertions/assertions_props`, `messages`, `filenames`, `lists`, `terms`, `system`, `pretty_print`, `vndict`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

13.2 Documentation on exports (autodoc)

autodoc/10: PREDICATE

Usage:
`autodoc(Format,Main,LibPaths,SysLibPaths,Idxs,Components,PathAliasF,StartPage,PaperTy`

- *Description:* This predicate is the main interface to the automatic documentation library.

  `Main` is the name of the source file being documented, and `Components` is a list of other files to be documented in relation to this one (see below). The output is a file whose contents document the main file, based on any assertions present in that file. The documentation is produced in the format given by `Format` (the name of the output file also depends on `Format`). The formats supported are given by `supported_format/1` in library `autodocformats`.

  If the manual should include other files (normally as chapters) `Components` is nonempty and contains the complete names of the component files. These files will appear in the manual in the order in which they appear in `Components`. These files can be written manually or generated automatically, but must be in a format compatible with `Format`. In particular, they can be also generated automatically with this same predicate by simply calling with `'-component'` as one of the options in `Options`.

`LibPaths` is a list of library paths where files used by the module being documented may be found. `SysLibPaths` is similar to `LibPaths` but provides paths to *system* libraries. `PathAliasF` is the name of a module containing path aliases. `Idxs` is a list of index names (the indices generated automatically). `StartPage` is 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.

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

`Format` is a supported typesetting format.	(`supported_format/1`)
`Main` is the name of a file.	(`filename/1`)
`LibPaths` is a list of `atm`s.	(`list/2`)
`SysLibPaths` is a list of `atm`s.	(`list/2`)
`Idxs` is a list of `atm`s.	(`list/2`)
`Components` is a list of `filename`s.	(`list/2`)
`PathAliasF` is the name of a file.	(`filename/1`)
`StartPage` is an integer.	(`int/1`)
`PaperType` is an atom.	(`atm/1`)
`Opts` is a list of `miscopt`s.	(`list/2`)

generate_man_page/5: PREDICATE

Usage: `generate_man_page(Main,LibPaths,SysLibPaths,PathAliasF,Opts)`

– *Description:* Generates a brief description of the application or library in unix man format. `Main` is the name of a the source file which is the main file of the application. `LibPaths` is a list of library paths the module being documented may be found. `SysLibPaths` is similar to `LibPaths` but provides paths to *system* libraries. `PathAliasF` is the name of a module containing path aliases.

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

`Main` is the name of a file.	(`filename/1`)
`LibPaths` is a list of `atm`s.	(`list/2`)
`SysLibPaths` is a list of `atm`s.	(`list/2`)
`PathAliasF` is an atom.	(`atm/1`)
`Opts` is a list of `miscopt`s.	(`list/2`)

generate_description/6: PREDICATE

Usage:
`generate_description(Format,Main,LibPaths,SysLibPaths,PathAliasF,Opts)`

– *Description:* Generates a brief description of the application or library in a file. This file is intended for inclusion in a larger file that is a catalog of aplications or libraries. The file is produced in the format given by `Format`. `Main` is the name of a the source file which is the main file of the application. The name of the output file depends on `Format` – see see `supported_format/1` in library `autodocformats`. `LibPaths` is a list of library paths the module being documented may use. `SysLibPaths` is similar to `LibPaths` but provides paths to *system* libraries. `PathAliasF` is the name of a module containing path aliases.

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

`Format` is a supported typesetting format.	(`supported_format/1`)

`Main` is the name of a file. (`filename/1`)
`LibPaths` is a list of `atm`s. (`list/2`)
`SysLibPaths` is a list of `atm`s. (`list/2`)
`PathAliasF` is the name of a file. (`filename/1`)
`Opts` is a list of `miscopts`. (`list/2`)

generate_info_dir_entry/5: PREDICATE

Usage: `generate_info_dir_entry(Main,LibPaths,SysLibPaths,PathAliasF,Opts)`

- *Description:* Generates a one line description of the application or library in a file. This file is intended for inclusion in a larger file that is a directory of `emacs info` manuals. The file is produced in ascii. `Main` is the name of a the source file which is the main file of the application. The name of the output file is `Main.infoindex`. `LibPaths` is a list of library paths the module being documented may use. `SysLibPaths` is similar to `LibPaths` but provides paths to *system* libraries. `PathAliasF` is the name of a module containing path aliases.
- *The following properties should hold at call time:*
  `Main` is the name of a file. (`filename/1`)
  `LibPaths` is a list of `atm`s. (`list/2`)
  `SysLibPaths` is a list of `atm`s. (`list/2`)
  `PathAliasF` is an atom. (`atm/1`)
  `Opts` is a list of `miscopts`. (`list/2`)

rewrite_docstring/4: PREDICATE

Usage: `rewrite_docstring(Format,Idxs,S,RS)`

- *Description:* Rewrites a documentation string `S` into another one `RS`, while processing any embedded commands, processing some directly and converting others into the appropriate commands for output format `Format`. Also, eliminates any blanks or tabs that appear at the beginning of a line. This is needed for example in `texinfo`: although leading blanks are OK for the printed manuals, they produce weird info files.
- *The following properties should hold at call time:*
  `Format` is a supported typesetting format. (`supported_format/1`)
  `Idxs` is a list of `atm`s. (`list/2`)
  `S` is a *documentation string.* (`docstring/1`)
- *The following properties should hold upon exit:*
  `RS` is a *documentation string.* (`docstring/1`)

modtype/1: REGTYPE

```
modtype(application).
modtype(use_module).
modtype(include).
modtype(part).
```

Usage:

- *Description:* Represents the type of file being documented.

index_comment/2: (UNDOC_REEXPORT)

Imported from `autodocformats` (see the corresponding documentation for details).

option_comment/2: (UNDOC_REEXPORT)

Imported from `autodocformats` (see the corresponding documentation for details).

13.3 Documentation on multifiles (`autodoc`)

library_directory/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

The predicate is of type *dynamic*.

13.4 Version/Change Log (`autodoc`)

Version 0.0 (1996/10/10)

First prototype. (Manuel Hermenegildo)

14 Low level documentation format definitions

Author(s): Manuel Hermenegildo.

Version: 1.9#58 (2002/4/19, 20:59:33 CEST)

Version of last change: 1.9#50 (2000/4/18, 3:22:13 CEST)

This module defines the interface for the auxiliary predicates needed by the automatic documentation generation library: these predicates determine the precise format in which the output is written.

Also, this module includes definitions of these predicates for a few formats. The main output format supported is `texinfo` (see The GNU Texinfo Documentation System manual for more info), from which printed manuals and several other printing and on-line formats can be easily generated automatically (including info, html, etc.). There is also some limited support for direct output in unix `man` format and direct `html` (but note that html can also be generated in a better way by first generating texinfo and then using one of the available converters). For texinfo, the documentation for a module is a texinfo chapter, suitable for inclusion in a wrapper “main” document file. A simple example of the use of this library for generating a texinfo reference manual (including a driver script, useful Makefiles, etc.) is included with the library source code. Other examples can be found in the Ciao documentation directory (i.e., the Ciao manuals themselves).

14.1 Usage and interface (`autodocformats`)

- **Library usage:**
  `:- use_module(library(autodocformats)).`
- **Exports:**
  - *Predicates:*
    `supported_format_suffix/2`, `index_comment/2`, `option_comment/2`, `format_front_matter/19`, `format_intro/10`, `format_module_usage/14`, `format_predicates_begin/4`, `format_predicate_begin/6`, `format_predicate_comment/3`, `format_predicate_end/2`, `format_native_declaration/3`, `format_predicates_end/2`, `format_multiple_usage_header/3`, `format_usage_header/2`, `format_head_descriptor/5`, `format_other_assrt_header/2`, `format_site_begin/4`, `format_site_end/2`, `format_properties_begin/2`, `format_property/7`, `format_properties_end/2`, `format_description/3`, `format_other_info/10`, `format_includes_and_end_matter/6`, `verbatimize_string/3`, `rewrite_command/4`.
  - *Regular Types:*
    `supported_format/1`.
- **Other modules used:**
  - *Application modules:*
    `lpdoclib(autodoc)`, `lpdoclib(comments)`.
  - *System library modules:*
    `aggregates`, `format`, `write`, `streams`, `errhandle`, `operators`, `messages`, `lists`, `terms`, `assertions/assertions_props`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

14.2 Documentation on exports (`autodocformats`)

supported_format/1: REGTYPE

Usage: `supported_format(Format)`

- *Description:* `Format` is a supported typesetting format.

supported_format_suffix/2: PREDICATE

Usage: `supported_format_suffix(Format,Suffix)`

- *Description:* `Format` is a format for which formatting code is available. `Suffix` is the suffix that should be used for the generated files.
- *The following properties should hold upon exit:*
  `Format` is a supported typesetting format. (`supported_format/1`)
  `Suffix` is currently instantiated to an atom. (`atom/1`)

index_comment/2: PREDICATE

Usage: `index_comment(Index,Text)`

- *Description:* `Type` is a type of index which is supported. `Text` describes the index contents.
- *The following properties should hold upon exit:*

  `Index` is currently instantiated to an atom. (`atom/1`)

  `Text` is a string (a list of character codes). (`string/1`)

option_comment/2: PREDICATE

Usage: `option_comment(Option,Text)`

- *Description:* `Option` is a documentation option which is supported. `Text` describes the effect of selecting that option. Currently supported options are:

```
option_comment('-v',"Verbose output (good for debugging).        ").
option_comment('-nobugs',"Do not include information on bugs.         "
option_comment('-noauthors',"Do not include author names.
option_comment('-noversion',"Do not include version information.
option_comment('-nochangelog',"Do not include change log.
option_comment('-nopatches',"Do not include comments for patches.
option_comment('-modes',"Do not translate modes and their arguments
                               (except for properties)
option_comment('-headprops',"Do not move head properties to body.
option_comment('-literalprops',"Do not use text to document properties.
option_comment('-nopropnames',"Do not include property names in prop te
option_comment('-noundefined',"Do not signal undefined properties in te
option_comment('-nopropsepln',"Do not put each property in a separate l
option_comment('-norefs',"Do not include a 'References' appendix.     "
option_comment('-nobullet',"Do not generate initial bullet index
                               (.htmlbullet) with .htmlindex file. Sel
                               only one manual will be installed in DO
option_comment('-nosysmods',"Do not include system modules in list of
                               libraries used.").
option_comment('-noengmods',"Do not include system engine modules in li
                               of libraries used.").
option_comment('-noisoline',"Do not include *textual* description that
                               given usage conforms to the ISO standar
option_comment('-propmods',"Include module name to which props belong."
option_comment('-shorttoc',"Produce shorter table of contents (no entri
                               for individual defs of preds, props, et
option_comment('-regtypeprops',"Include in the doc for regtypes the glo
                               prop stating that they are indeed regty
option_comment('-onesided',"For printing on one side (default is two)."
```

- *The following properties should hold upon exit:*

  `Option` is a supported documentation option. (`supported_option/1`)

  `Text` is a string (a list of character codes). (`string/1`)

format_front_matter/19: PREDICATE

`format_front_`
`matter(Format,ModuleType,MainOrComp,Name,NDName,Version,GVers,Title,Authors,Subtitle`

This predicate defines the first part of the format of the main file of a manual. `Format` is the type of output (e.g., texinfo, latex, etc.) – different clauses of the predicate can be defined for different formats. `Name` is the name of the application (taken from the name of the input file). `NDName` is the same, but without `_doc`, if applicable. `Version` is the version of the first `comment/2` entry which specifies a version number (which should be the current version). This is the version of the last local change. `GVers` is the global version. `Title` is the intended title of the application (taken from the approriate `comment/2` declaration). `Authors` is a (possibly empty) list of author names. `Subtitle` is a (possibly empty) list of subtitle (e.g., address, explanation) lines. `Copyright` is the copyright text (taken from the appropriate `comment/2` declaration). `Summary` is a brief summary of the contents of the manual (taken from the appropriate `comment/2` declaration). `Indices` is a list of index names (the indices to be generated). `StartPage` is the page number of the first page of the manual. `I` and `O` are the names of the input and output files being processed (which can be used for example to put a comment in the generated file saying that the file has been generated automatically from these files and therefore should probably not be edited by hand). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_front_`
`matter(Format,ModuleType,MainOrComp,Name,NDName,Version,GVers,Title,Authors,Subtitle`

- *Description:* A texinfo main file is generated. `Name` is used as the name of the file. `Title` is used as the title of the document. The `Version`, `Authors`, and the `Subtitle` will go in the title page. `Copyright` goes in the back of the title page.
- *The following properties should hold at call time:*

`Format` and `texinfo` unify.	(`= /2`)
`Name` is currently instantiated to an atom.	(`atom/1`)
`NDName` is currently instantiated to an atom.	(`atom/1`)
`Version` is a complete version descriptor.	(`version_descriptor/1`)
`Title` is a string (a list of character codes).	(`string/1`)
`Authors` is a list of `string`s.	(`list/2`)
`Subtitle` is a list of `string`s.	(`list/2`)
`Copyright` is a string (a list of character codes).	(`string/1`)
`Summary` is a string (a list of character codes).	(`string/1`)
`Indices` is a list of `atom`s.	(`list/2`)
`StartPage` is an integer.	(`int/1`)
`PaperType` is an atom.	(`atm/1`)
`Opts` is a list of `supported_options`.	(`list/2`)
`I` is the name of a file.	(`filename/1`)
`O` is the name of a file.	(`filename/1`)
`OS` is an open stream.	(`stream/1`)

format_intro/10: PREDICATE

`format_`
`intro(Format,FileType,ModuleType,Name,NDName,Summary,Version,Comment,OS,IntroOS)`

This predicate defines the format of the introduction and some auxiliary information. `Format` is the type of output (e.g., texinfo, latex, etc.). `FileType` is the type of file (main, component, etc.). `ModuleType` is how the module is used (use_module, etc.). `Name` is the module/application name. `NDName` is the same, but without `_doc`, if applicable. `Summary` is a brief summary of the contents of the manual (taken from the appropriate `comment/2` declaration, if available). `Version` is the version of the first `comment/2` entry which specifies a version number (which should be the current version). `Comment` is the introductory text for the module (taken from the `comment/2` declaration, if available). `OS` is the output stream to which writes should be made (normally points to the output file). `IntroOS` is the output stream for the introduction.

Usage: `format_intro(Format,FileType,ModuleType,Name,NDName,Summary,Version,Comment,OS,IntroOS)`

- *Description:* Formats intro stuff: summary, intro, etc.
- *The following properties should hold at call time:*

`Format` and `texinfo` unify.	(`= /2`)
`FileType` is any term.	(`term/1`)
`ModuleType` is currently instantiated to an atom.	(`atom/1`)
`Name` is a string (a list of character codes).	(`string/1`)
`NDName` is a string (a list of character codes).	(`string/1`)
`Summary` is a string (a list of character codes).	(`string/1`)
`Version` is a complete version descriptor.	(`version_descriptor/1`)
`Comment` is a string (a list of character codes).	(`string/1`)
`OS` is an open stream.	(`stream/1`)
`IntroOS` is an open stream.	(`stream/1`)

format_module_usage/14: PREDICATE

`format_module_usage(Format,Name,Type,Exports,Mults,UMods,IUMods,SMods,EMOds,Ops,NDecls,NModes,Indi`

This predicate defines the format of the usage info for the module. `Format` is the type of output (e.g., texinfo, latex, etc.). `Name` is the name of the module (taken from the `module/2` declaration). `Exports` contains the predicates exported by the module (taken from the `module/2` declaration). `UMods` contains the user modules imported by the module. `SMods` contains the system modules imported by the module. `EMods` contains the internal (engine) modules imported by the module. `Ops` contains any exported operator definitions. `NDecls` contains any exported new declarations. `NModes` contains any new mode definitions. `Indices` is a list of index names (the indices to be generated). `OS` is the output stream to which writes should be made (normally points to the output file).

format_predicates_begin/4: PREDICATE

`format_predicates_begin(Format,Name,Text,OS)`

This predicate defines the format of the first part of the documentation for a set of predicates. `Format` is the type of output (e.g., texinfo, latex, etc.). `Name` is the module name. `Text` describes which set of predicates is being documented (e.g., exported predicates, imported predicates, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_predicates_begin(Format,Name,Text,OS)`

- *Description:* Predicates will formatted as an itemized list. Corresponds to a begin itemize.

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

  `Format` and `texinfo` unify. (`= /2`)

  `Name` is currently instantiated to an atom. (`atom/1`)

  `Text` is a string (a list of character codes). (`string/1`)

  `OS` is an open stream. (`stream/1`)

format_predicate_begin/6: PREDICATE

`format_predicate_begin(Format,Type,Pred,Indices,Opts,OS)`

This predicate defines the format of the first part of the documentation for a predicate (the header). `Format` is the type of output (e.g., texinfo, latex, etc.). `Type` is the type of predicate being documented (predicate, property, type, etc.). `Pred` is the name of the predicate. `Indices` is a list of index names (the indices to be generated). `Opts` are the formatting options. `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_predicate_begin(Format,Type,Pred,Indices,Opts,OS)`

- *Description:* Each predicate is an item in an itemized list.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Type` is currently instantiated to an atom. (`atom/1`)

  `Pred` is a Name/Arity structure denoting a predicate name:

  ```
  predname(P/A) :-
          atm(P),
          nnegint(A).
  ```

  (`predname/1`)

  `Indices` is a list of `atoms`. (`list/2`)

  `Opts` is a list of `supported_options`. (`list/2`)

  `OS` is an open stream. (`stream/1`)

format_predicate_comment/3: PREDICATE

`format_predicate_comment(Format,Comment,OS)`

This predicate defines the format of `Comment`, an introductory text for the predicate (taken from the `comment/2` declaration, if available). `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_predicate_comment(Format,Comment,OS)`

- *Description:* Predicate comments are output as is (in a separate paragraph).
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Comment` is a string (a list of character codes). (`string/1`)

  `OS` is an open stream. (`stream/1`)

format_predicate_end/2: PREDICATE

`format_predicate_end(Format,OS)`

This predicate defines the format of the last part of the documentation for a predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_predicate_end(Format,OS)`

- *Description:* Noop (each predicate is an item in an itemized list).
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_native_declaration/3: PREDICATE

Usage: `format_native_declaration(Format,Decl,OS)`

- *Description:* This predicate defines the format for documenting miscellaneous declarations such as `meta_predicate/1`, `multifile/1`, `dynamic/1`, `data/1`, etc. `Format` is the type of output (e.g., texinfo, latex, etc.). `Decl` contains the declaration info. `OS` is the output stream to which writes should be made (normally points to the output file). No special comment is generated for the default value (`static`).
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Decl` is any term. (`term/1`)

  `OS` is an open stream. (`stream/1`)

format_predicates_end/2: PREDICATE

`format_predicates_end(Format,OS)`

This predicate defines the format of the last part of the documentation for a set of predicates. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_predicates_end(Format,OS)`

- *Description:* Predicates will formatted as an itemized list. Corresponds to an end itemize.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_multiple_usage_header/3: PREDICATE

`format_multiple_usage_header(Format,N,OS)`

This predicate defines the format of the header describing each usage for a predicate with multiple declarations. `Format` is the type of output (e.g., texinfo, latex, etc.). `N` is the usage number. `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_multiple_usage_header(Format,N,OS)`

- *Description:* Usage header in bold in separate paragraph.

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

  `Format` and `texinfo` unify. (`= /2`)

  `N` is currently instantiated to an integer. (`integer/1`)

  `OS` is an open stream. (`stream/1`)

format_usage_header/2: PREDICATE

`format_usage_header(Format,OS)`

This predicate defines the format of the header describing a single usage for a predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_usage_header(Format,OS)`

- *Description:* Usage header in bold in separate paragraph.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_head_descriptor/5: PREDICATE

`format_head_descriptor(Format,HD,Type,Standard,OS)`

This predicate defines the format of a predicate descriptor. `Format` is the type of output (e.g., texinfo, latex, etc.). `HD` describes the head of the predicate (as a structure whose argunents are the variable names). `Type` is the type of predicate (predicate, function, declaration, ...). `Standard` contains the atom `iso` is the predicate is iso-compliant. `OS` is the output stream to which writes should be made (normally points to the output file.

Usage: `format_head_descriptor(Format,HD,Type,Standard,OS)`

- *Description:* Head descriptors are formatted literally, in `@code` format. ISO compliance is formatted using the `@key` command.
- *The following properties should hold at call time:*

  `Format` is currently instantiated to an atom. (`atom/1`)

  `HD` is any term. (`term/1`)

  `Type` is currently instantiated to an atom. (`atom/1`)

  `Standard` is currently instantiated to an atom. (`atom/1`)

  `OS` is an open stream. (`stream/1`)

format_other_assrt_header/2: PREDICATE

`format_other_assrt_header(Format,OS)`

This predicate defines the format of the header general properties of a predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_other_assrt_header(Format,OS)`

- *Description:* Header in bold in separate paragraph.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_site_begin/4: PREDICATE

`format_site_begin(Format,Text,Bullet,OS)`

This predicate defines the format of the header of the description of a set of properties for a given *site* (at calls, on success, global properties, etc.). `Format` is the type of output (e.g., texinfo, latex, etc.). `Text` describes the site. `Bullet` says if a bullet should precede the text. `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_site_begin(Format,Text,Bullet,OS)`

- *Description:* Each site is an item in an itemized list.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Text` is a string (a list of character codes). (`string/1`)

  `Bullet` is an atom. (`atm/1`)

  `OS` is an open stream. (`stream/1`)

format_site_end/2: PREDICATE

`format_site_end(Format,OS)`

This predicate defines the format of the end of the description of a set of properties for a given site. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_site_end(Format,OS)`

- *Description:* Simply end of item (a newline).
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_properties_begin/2: PREDICATE

`format_properties_begin(Format,OS)`

This predicate defines the beginning of the description of the properties for a given predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `N` is the usage number. `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_properties_begin(Format,OS)`

- *Description:* Properties are formatted as a (second level) itemized list.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `OS` is an open stream. (`stream/1`)

format_property/7: PREDICATE

`format_property(Format,Prop,PM,DocString,VarNames,Opts,OS)`

This predicate defines the formatting for a property. `Format` is the type of output (e.g., texinfo, latex, etc.). `Prop` is the actual property (a term). `PM` is the module in which the property is defined. `VarNames` is a list of (possibly repeated) variable names, corresponding to the names in the head pattern. `DocString` contains `~s` in the places where the variables names should appear. Note that this is suitable for use as arguments for a call to `format/2`.

`OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_property(Format,Prop,PM,DocString,VarNames,Opts,OS)`

- *Description:* Properties are formatted as running text.
- *The following properties should hold at call time:*

`Format` is an atom.	(`atm/1`)
`Prop` is any term.	(`term/1`)
`PM` is an atom.	(`atm/1`)
`DocString` is a string (a list of character codes).	(`string/1`)
`VarNames` is a list of `strings`.	(`list/2`)
`Opts` is a list of `supported_options`.	(`list/2`)
`OS` is an open stream.	(`stream/1`)

format_properties_end/2: PREDICATE

`format_properties_end(Format,OS)`

This predicate defines the end of the description of the properties for a given predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `N` is the usage number. `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_properties_end(Format,OS)`

- *Description:* End of the (second level) itemized list.
- *The following properties should hold at call time:*

`Format` and `texinfo` unify.	(`= /2`)
`OS` is an open stream.	(`stream/1`)

format_description/3: PREDICATE

`format_description(Format,Desc,OS)`

This predicate defines the format of `Desc`, an introductory text for the predicate (taken from the comment part of a `pred/1` declaration, if available). These coments are generally used to describe a particular usage of the predicate. `Format` is the type of output (e.g., texinfo, latex, etc.). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage: `format_description(Format,Desc,OS)`

- *Description:* The description of the usage is a separate item in the site item list.
- *The following properties should hold at call time:*

`Format` is currently instantiated to an atom.	(`atom/1`)
`Desc` is a string (a list of character codes).	(`string/1`)
`OS` is an open stream.	(`stream/1`)

format_other_info/10: PREDICATE

No further documentation available for this predicate.

format_includes_and_end_matter/6: PREDICATE

`format_includes_and_end_matter(Format,Name,Components,Indices,Opts,OS)`

This predicate generates code for including the components of a complex document if needed, and produces the final matter of the main file. `Format` is the type of output (e.g., texinfo, latex, etc.) – different clauses of the predicate can be defined for different formats. `Name` is the name of the application (taken from the name of the input file). If the manual should include other files (e.g., as chapters) `Components` is nonempty and contains the complete names of the component files. These files will appear in the manual in the order in which they appear in `Components`. These files can be written manually or generated automatically by `document_module/2` or any other means, but must be in a format compatible with `Format`. `Indices` contains the index names (the indices to be generated). `OS` is the output stream to which writes should be made (normally points to the output file).

Usage:
`format_includes_and_end_matter(Format,Name,Components,Indices,Opts,OS)`

- *Description:* The components in `Components`, if any, are included as chapters. `Name` is used as the basename of the index component files. If no components, then everything should go in-line (including indices).
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Name` is the name of a file. (`filename/1`)

  `Components` is a list of `filename`s. (`list/2`)

  `Indices` is a list of `atom`s. (`list/2`)

  `Opts` is a list of `atom`s. (`list/2`)

  `OS` is an open stream. (`stream/1`)

verbatimize_string/3: PREDICATE

Usage:

- *Description:* Escapes needed characters in string as needed for the verbatim command supported natively by the format.
- *The following properties should hold at call time:*

  `Format` is a supported typesetting format. (`supported_format/1`)

  `Arg2` is a string (a list of character codes). (`string/1`)

  `Arg3` is any term. (`term/1`)
- *The following properties should hold upon exit:*

  `Format` is a supported typesetting format. (`supported_format/1`)

  `Arg2` is a string (a list of character codes). (`string/1`)

  `Arg3` is a string (a list of character codes). (`string/1`)

rewrite_command/4: PREDICATE

`rewrite_command(Format,Command,Indices,NewCommand)`

Defines the translation between the special commands which can appear in strings inside assertions and the formatting target. `Indices` is a list of index names (the indices to be generated).

Usage 1: `rewrite_command(Format,Command,Indices,NewCommand)`

- *Description:* Defines the translation between the special commands which can appear in strings inside assertions and texinfo.
- *The following properties should hold at call time:*

  `Format` and `texinfo` unify. (`= /2`)

  `Command` is a structure denoting a command that is admissible in strings inside assertions. (`stringcommand/1`)

  `Indices` is a list of `atoms`. (`list/2`)
- *The following properties should hold upon exit:*

  `NewCommand` is a string (a list of character codes). (`string/1`)

Usage 2: `rewrite_command(Format,Command,Indices,NewCommand)`
- *Description:* Defines the translation between the special commands which can appear in strings inside assertions and html. This is still somewhat incomplete.
- *The following properties should hold at call time:*

  `Format` and `html` unify. (`= /2`)

  `Command` is a structure denoting a command that is admissible in strings inside assertions. (`stringcommand/1`)

  `Indices` is a list of `atoms`. (`list/2`)
- *The following properties should hold upon exit:*

  `NewCommand` is a string (a list of character codes). (`string/1`)

Usage 3: `rewrite_command(Format,Command,Indices,NewCommand)`
- *Description:* Defines the translation between the special commands which can appear in strings inside assertions and man.
- *The following properties should hold at call time:*

  `Format` and `man` unify. (`= /2`)

  `Command` is a structure denoting a command that is admissible in strings inside assertions. (`stringcommand/1`)

  `Indices` is a list of `atoms`. (`list/2`)
- *The following properties should hold upon exit:*

  `NewCommand` is a string (a list of character codes). (`string/1`)

Usage 4: `rewrite_command(Format,Command,Indices,NewCommand)`
- *Description:* Defines the translation between the special commands which can appear in strings inside assertions and ascii.
- *The following properties should hold at call time:*

  `Format` and `ascii` unify. (`= /2`)

  `Command` is a structure denoting a command that is admissible in strings inside assertions. (`stringcommand/1`)

  `Indices` is a list of `atoms`. (`list/2`)
- *The following properties should hold upon exit:*

  `NewCommand` is a string (a list of character codes). (`string/1`)

14.3 Version/Change Log (`autodocformats`)

Version 0.0 (1996/10/10)
First prototype.

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.

[DL93] S.K. Debray and N.W. Lin.
Cost analysis of logic programs.
ACM Transactions on Programming Languages and Systems, 15(5):826–875, November 1993.

[DLGH97] S.K. Debray, P. L\'opez-Garc\'\ia, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL97]
S.K. Debray, P. L\'opez-Garc\'\ia, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[Her99] M. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
Technical Report CLIP10/99.0, Facultad de Inform\'atica, UPM, September 1999.

[Her00] M. Hermenegildo.
A Documentation Generator for (C)LP Systems.
In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[JL88] D. Jacobs and A. Langen.
Compilation of Logic Programs for Restricted And-Parallelism.
In *European Symposium on Programming*, pages 284–297, 1988.

[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.

[LGHD96] P. L\'opez-Garc\'\ia, M. Hermenegildo, and S.K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 22:715–734, 1996.

[MH89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[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

Property Definition Index

Regular Type Definition Index

Mode Definition Index

Concept Definition Index

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.