The lpdoc Documentation Generator

An Automatic Documentation Generator for (C)LP Systems
REFERENCE MANUAL
The Ciao Documentation Series
`http://www.ciaohome.org/`
Generated/Printed on: 1 April 2011
Technical Report CLIP 5/97.1-<*version below*>

Manuel Hermenegildo
José Francisco Morales

Table of Contents

Summary 1

1 Introduction 3
- 1.1 Overview of this document 3
- 1.2 lpdoc operation - source and target files 3
- 1.3 lpdoc usage 5

PART I - LPdoc Reference Manual 7

2 Generating Installing and Accessing Manuals . . . 9
- 2.1 Generating a manual from the Ciao Emacs mode 9
- 2.2 Generating a manual 9
- 2.3 Working on a manual 11
- 2.4 Cleaning up the documentation directory 11
- 2.5 Installing a generated manual in a public area 12
- 2.6 Enhancing the documentation being generated 12
- 2.7 Accessing on-line manuals 13
  - 2.7.1 Accessing html manuals 13
  - 2.7.2 Accessing info manuals 13
  - 2.7.3 Accessing man manuals 14
  - 2.7.4 Putting it all together 15
- 2.8 Some usage tips 15
  - 2.8.1 Ensuring Compatibility with All Supported Target Formats 15
  - 2.8.2 Writing comments which document version/patch changes 15
  - 2.8.3 Documenting Libraries and/or Applications 16
  - 2.8.4 Documenting files which are not modules 16
  - 2.8.5 Splitting large documents into parts 16
  - 2.8.6 Documenting reexported predicates 17
  - 2.8.7 Separating the documentation from the source file 17
  - 2.8.8 Generating auxiliary files (e.g. READMEs) 17
- 2.9 Troubleshooting 18

3 LPdoc Comments and Mark-up Language 19
- 3.1 Usage and interface (`comments`) 19
- 3.2 Documentation on exports (`comments`) 19
  - docstring/1 (prop) 19
  - stringcommand/1 (prop) 20
  - version_descriptor/1 (regtype) 25
  - filetype/1 (regtype) 26
  - doc_id_type/3 (pred) 26
- 3.3 Documentation on internals (`comments`) 26
  - doc/2 (decl) 26
  - version_number/1 (regtype) 32
  - ymd_date/1 (regtype) 32
  - time_struct/1 (regtype) 32
  - version_maintenance_type/1 (regtype) 32

4 The Ciao assertion package 35
4.1 More info 35
4.2 Some attention points 35
4.3 Usage and interface (`assertions_doc`) 36
4.4 Documentation on new declarations (`assertions_doc`) 36
(pred)/1 (decl) 36
(pred)/2 (decl) 37
(texec)/1 (decl) 37
(texec)/2 (decl) 37
(calls)/1 (decl) 37
(calls)/2 (decl) 37
(success)/1 (decl) 38
(success)/2 (decl) 38
(test)/1 (decl) 38
(test)/2 (decl) 38
(comp)/1 (decl) 39
(comp)/2 (decl) 39
(prop)/1 (decl) 39
(prop)/2 (decl) 40
(entry)/1 (decl) 40
(exit)/1 (decl) 40
(exit)/2 (decl) 41
(modedef)/1 (decl) 41
(decl)/1 (decl) 41
(decl)/2 (decl) 41
doc/2 (decl) 41
comment/2 (decl) 42
4.5 Documentation on exports (`assertions_doc`) 42
check/1 (pred) 42
trust/1 (pred) 42
true/1 (pred) 43
false/1 (pred) 43

5 Types and properties related to assertions 45
5.1 Usage and interface (`assertions_props`) 45
5.2 Documentation on exports (`assertions_props`) 45
assrt_body/1 (regtype) 45
head_pattern/1 (prop) 46
complex_arg_property/1 (regtype) 46
property_conjunction/1 (regtype) 47
property_starterm/1 (regtype) 47
complex_goal_property/1 (regtype) 47
nabody/1 (prop) 48
dictionary/1 (regtype) 48
c_assrt_body/1 (regtype) 48
s_assrt_body/1 (regtype) 48
g_assrt_body/1 (regtype) 49
assrt_status/1 (regtype) 50
assrt_type/1 (regtype) 50
predfunctor/1 (regtype) 50
propfunctor/1 (regtype) 50
docstring/1 (prop) 50

6 Declaring regular types 51
6.1 Defining properties 51
6.2 Usage and interface (regtypes_doc) 53
6.3 Documentation on new declarations (regtypes_doc) 54
(regtype)/1 (decl) 54
(regtype)/2 (decl) 54

7 Basic data types and properties 55
7.1 Usage and interface (basic_props) 55
7.2 Documentation on exports (basic_props) 55
term/1 (regtype) 55
int/1 (regtype) 56
nnegint/1 (regtype) 56
flt/1 (regtype) 57
num/1 (regtype) 57
atm/1 (regtype) 58
struct/1 (regtype) 58
gnd/1 (regtype) 59
gndstr/1 (regtype) 59
constant/1 (regtype) 60
callable/1 (regtype) 60
operator_specifier/1 (regtype) 61
list/1 (regtype) 61
list/2 (regtype) 62
nlist/2 (regtype) 62
member/2 (prop) 63
sequence/2 (regtype) 63
sequence_or_list/2 (regtype) 64
character_code/1 (regtype) 64
string/1 (regtype) 65
num_code/1 (regtype) 65
predname/1 (regtype) 65
atm_or_atm_list/1 (regtype) 66
compat/2 (prop) 66
inst/2 (prop) 66
iso/1 (prop) 67
deprecated/1 (prop) 67
not_further_inst/2 (prop) 67
sideff/2 (prop) 68
(regtype)/1 (prop) 68
native/1 (prop) 68
native/2 (prop) 68
no_rtcheck/1 (prop) 68
eval/1 (prop) 69
equiv/2 (prop) 69
bind_ins/1 (prop) 69
error_free/1 (prop) 69
memo/1 (prop) 69
filter/2 (prop) 69
flag_values/1 (regtype) 69
pe_type/1 (prop) 70

8 Properties which are native to analyzers 71
8.1 Usage and interface (native_props) 71
8.2 Documentation on exports (native_props) 71
clique/1 (prop) 71
clique_1/1 (prop) 72
constraint/1 (prop) 72
covered/1 (prop) 72
covered/2 (prop) 72
exception/1 (prop) 72
exception/2 (prop) 72
fails/1 (prop) 73
finite_solutions/1 (prop) 73
have_choicepoints/1 (prop) 73
indep/1 (prop) 73
indep/2 (prop) 73
is_det/1 (prop) 73
linear/1 (prop) 74
mshare/1 (prop) 74
mut_exclusive/1 (prop) 74
no_choicepoints/1 (prop) 74
no_exception/1 (prop) 75
no_exception/2 (prop) 75
no_signal/1 (prop) 75
no_signal/2 (prop) 75
non_det/1 (prop) 75
nonground/1 (prop) 75
not_covered/1 (prop) 75
not_fails/1 (prop) 75
not_mut_exclusive/1 (prop) 76
num_solutions/2 (prop) 76
solutions/2 (prop) 76
possibly_fails/1 (prop) 76
possibly_nondet/1 (prop) 77
relations/2 (prop) 77
sideff_hard/1 (prop) 77
sideff_pure/1 (prop) 77
sideff_soft/1 (prop) 77
signal/1 (prop) 77
signal/2 (prop) 77
signals/2 (prop) 78
size/2 (prop) 78
size/3 (prop) 78
size_lb/2 (prop) 78
size_o/2 (prop) 78
size_ub/2 (prop) 78
size_metric/3 (prop) 78
size_metric/4 (prop) 78
steps/2 (prop) 79
steps_lb/2 (prop) 79
steps_o/2 (prop) 79
steps_ub/2 (prop) 79
tau/1 (prop) 79
terminates/1 (prop) 80
test_type/2 (prop) 80
throws/2 (prop) 80
user_output/2 (prop) 80

intervals/2 (prop) . . . 80
instance/2 (prop) . . . 80

9 Meta-properties . . . 81
9.1 Usage and interface (meta_props) . . . 81
9.2 Documentation on exports (meta_props) . . . 81
call/2 (prop) . . . 81
(prop)/2 (prop) . . . 81
(regtype)/2 (prop) . . . 82
9.3 Documentation on multifiles (meta_props) . . . 82
callme/2 (pred) . . . 82
9.4 Documentation on internals (meta_props) . . . 82
prop_abs/1 (prop) . . . 82

10 An Example - Documenting a Library Module . . . 83

11 Auto Documenter Output for the Example Module . . . 89
11.1 Usage and interface (example_module) . . . 89
11.2 Documentation on exports (example_module) . . . 89
bar/1 (regtype) . . . 89
baz/1 (regtype) . . . 89
aorb/1 (regtype) . . . 89
tree_of/2 (regtype) . . . 90
list_or_aorb/2 (regtype) . . . 90
q/2 (pred) . . . 90
r/1 (pred) . . . 90
p/1 (pred) . . . 91
p/5 (pred) . . . 91
u/3 (pred) . . . 91
long/1 (prop) . . . 91
w/1 (pred) . . . 92
mytype/1 (pred) . . . 92
t/5 (pred) . . . 92
s/1 (pred) . . . 92
q/1 (pred) . . . 93
list/1 (regtype) . . . 93
11.3 Documentation on multifiles (example_module) . . . 94
p/3 (pred) . . . 94
11.4 Documentation on internals (example_module) . . . 95
list/2 (regtype) . . . 95
og/2 (modedef) . . . 95
is/2 (pred) . . . 96

12 Run-time checking of assertions . . . 97
12.1 Usage and interface (rtchecks_doc) . . . 98

13 Unit Testing Library . . . 99
13.1 Additional notes . . . 99
13.2 Usage and interface (unittest_doc) . . . 100

14 Installing lpdoc 101
14.1 Installing the source distribution (lpdoc) 101
14.2 Other software packages required (lpdoc) 101

PART II - LPdoc Internals Manual 103

15 Documentation Generation Library 105
15.1 Usage and interface (autodoc) 106
15.2 Documentation on exports (autodoc) 106
backend_id/1 (regtype) 106
backend_ignores_components/1 (pred) 107
backend_alt_format/2 (pred) 107
top_suffix/2 (pred) 107
index_comment/2 (pred) 107
supported_option/1 (prop) 107
option_comment/2 (pred) 107
docstate/1 (regtype) 108
docstate_backend/2 (pred) 108
docstate_currmod/2 (pred) 108
docstate_set_currmod/3 (pred) 108
docstate_opts/2 (pred) 108
docstate_set_opts/3 (pred) 108
docstate_set_customdic/3 (pred) 109
docstate_inputfile/2 (pred) 109
doc_message/2 (pred) 109
doc_message/3 (pred) 109
doc_opt/2 (pred) 109
doc_customdic_lookup/3 (pred) 109
doc_customdic_replace/4 (pred) 109
doc_customdic_get/3 (pred) 109
doc_customdic_maybe_get/3 (pred) 109
doc_doing_mainmod/1 (pred) 109
doc_modname/2 (pred) 109
docstate_has_index/2 (pred) 109
all_indices/2 (pred) 110
get_doc/4 (pred) 110
modtype/1 (regtype) 110
reset_output_dir_db/0 (pred) 110
ensure_output_dir_prepared/2 (pred) 110
get_autodoc_opts/3 (pred) 110
autodoc_gen_doctree/5 (pred) 110
infodir_base/2 (pred) 111
autodoc_compute_grefs/3 (pred) 111
autodoc_translate_doctree/3 (pred) 111
autodoc_finish/1 (pred) 111
autodoc_gen_alternative/2 (pred) 111
version_patch/2 (pred) 111
version_date/2 (pred) 111
version_numstr/2 (pred) 111
15.3 Documentation on multifiles (autodoc) 112
autodoc_finish_hook/1 (pred) 112
autodoc_gen_alternative_hook/2 (pred) 112

16 Documentation Abstract Syntax Tree....... 113

16.1 Usage and interface (`autodoc_doctree`).................... 113

16.2 Documentation on exports (`autodoc_doctree`)............. 113

cmd_type/1 (pred)............................. 113

doctree/1 (regtype)............................. 113

doctree_is_empty/1 (pred)......................... 114

is_nonempty_doctree/1 (pred)..................... 114

empty_doctree/1 (pred)........................... 114

doclink/1 (regtype)............................. 114

doclabel/1 (regtype)............................. 114

doclink_at/2 (pred)............................. 114

doclink_is_local/1 (pred)......................... 114

doctokens/1 (regtype)........................... 114

section_prop/2 (pred)............................ 115

section_select_prop/3 (pred)....................... 115

doctree_save/2 (pred)............................ 115

doctree_restore/2 (pred).......................... 115

doctree_simplify/2 (pred)......................... 115

doctree_putvars/5 (pred).......................... 115

doctree_scan_and_save_refs/2 (pred)................ 115

doctree_prepare_refs_translate_and_write/3 (pred)... 115

doctree_to_rawtext/3 (pred)....................... 115

doctree_translate_and_write/3 (pred)............... 116

escape_string/4 (pred)........................... 116

insert_printtoc/3 (pred).......................... 116

16.3 Documentation on multifiles (`autodoc_doctree`)............ 116

autodoc_rw_command_hook/4 (pred)............... 116

autodoc_escape_string_hook/5 (pred)............... 116

17 Handling the Document Structure.......... 117

17.1 Usage and interface (`autodoc_structure`)................. 117

17.2 Documentation on exports (`autodoc_structure`)........... 117

docstr_node/4 (pred)............................ 117

clean_docstr/0 (pred)............................ 117

parse_structure/0 (pred).......................... 117

check_deprecated_settings/0 (pred)................. 117

parse_structure_/2 (pred)......................... 117

parse_structure_list/2 (pred)...................... 118

is_list/1 (pred).................................. 118

standalone_docstr/1 (pred)........................ 118

get_mainmod/1 (pred)........................... 118

get_mainmod_spec/1 (pred)...................... 118

all_component_specs/1 (pred)..................... 118

component_spec/1 (pred)........................ 118

18 Access to Default Settings 119
18.1 Usage and interface (`autodoc_settings`) 119
18.2 Documentation on exports (`autodoc_settings`) 119
lpdoc_option/1 (pred) 119
check_setting/1 (pred) 119
setting_value_or_default/2 (pred) 119
setting_value_or_default/3 (pred) 119
setting_value/2 (pred) 120
all_setting_values/2 (pred) 120
get_command_option/1 (pred) 120
requested_file_formats/1 (pred) 120
load_vpaths/0 (pred) 120

LPdoc Backends 121

19 Texinfo Backend 123
19.1 Usage and interface (`autodoc_texinfo`) 123
19.2 Documentation on multifiles (`autodoc_texinfo`) 123
autodoc_escape_string_hook/5 (pred) 123
autodoc_rw_command_hook/4 (pred) 123
autodoc_finish_hook/1 (pred) 123
autodoc_gen_alternative_hook/2 (pred) 124

20 HTML Backend 125
20.1 Usage and interface (`autodoc_html`) 125
20.2 Documentation on multifiles (`autodoc_html`) 125
autodoc_escape_string_hook/5 (pred) 125
autodoc_rw_command_hook/4 (pred) 125
autodoc_finish_hook/1 (pred) 125
autodoc_gen_alternative_hook/2 (pred) 126

21 Resource Handling for the HTML Backend . . 127
21.1 Usage and interface (`autodoc_html_resources`) 127
21.2 Documentation on exports (`autodoc_html_resources`) 127
prepare_web_skel/1 (pred) 127
prepare_mathjax/0 (pred) 127
using_mathjax/1 (pred) 127

22 Template Support for the HTML Backend . . 129
22.1 Usage and interface (`autodoc_html_template`) 129
22.2 Documentation on exports (`autodoc_html_template`) 129
img_url/2 (pred) 129
fmt_html_template/3 (pred) 129

23 Man Pages (man) Backend 131
23.1 Usage and interface (`autodoc_man`) 131
23.2 Documentation on multifiles (`autodoc_man`) 131
autodoc_rw_command_hook/4 (pred) 131
autodoc_finish_hook/1 (pred) 131
autodoc_gen_alternative_hook/2 (pred) 131

24 Filesystem Abstraction 133
24.1 Usage and interface (autodoc_filesystem) 133
24.2 Documentation on exports (autodoc_filesystem) 133
filename/1 (regtype) 133
basename/1 (regtype) 133
subtarget/1 (regtype) 133
file_format_name/2 (pred) 134
supported_file_format/1 (pred) 134
file_format_provided_by_backend/3 (pred) 134
clean_fs_db/0 (pred) 134
get_output_dir/2 (pred) 134
get_cache_dir/2 (pred) 134
ensure_output_dir/1 (pred) 134
ensure_cache_dir/1 (pred) 134
main_absfile_in_format/2 (pred) 134
main_absfile_for_subtarget/3 (pred) 135
absfile_for_aux/3 (pred) 135
absfile_for_subtarget/4 (pred) 135
main_output_name/1 (pred) 135
get_subbase/3 (pred) 135
absfile_to_relfile/3 (pred) 135
clean_all/0 (pred) 135
clean_docs_no_texi/0 (pred) 135
clean_all_temporal/0 (pred) 135
clean_intermediate/0 (pred) 136
clean_tex_intermediate/0 (pred) 136

25 Indexing Commands (Definition and Formatting) 137
25.1 Usage and interface (autodoc_index) 137
25.2 Documentation on exports (autodoc_index) 137
get_idxsub/2 (pred) 137
get_idxbase/3 (pred) 137
typeindex/5 (pred) 137
idx_get_indices/3 (pred) 138
is_index_cmd/1 (pred) 138
codetype/1 (pred) 138
normalize_index_cmd/3 (pred) 138
fmt_idx_env/7 (pred) 138
fmt_index/4 (pred) 138

26 Database of Documentation References 139
26.1 Usage and interface (autodoc_refsdb) 139
26.2 Documentation on exports (autodoc_refsdb) 139
labgen_init/1 (pred) 139
labgen_clean/1 (pred) 139
labgen_get/2 (pred) 139
refs_clean/1 (pred) 139
add_refs_entry/2 (pred) 140
save_refs/1 (pred) 140
refs_closure_entry/3 (pred) 140
refs_closure_clean/1 (pred) 140
compute_refs_and_biblio/1 (pred) 140
prepare_current_refs/1 (pred) 140
clean_current_refs/1 (pred) 140
secttree/1 (regtype) 140
secttree_resolve/3 (pred) 140
get_main_title/2 (pred) 141

27 Error Messages 143
27.1 Usage and interface (autodoc_errors) 143
27.2 Documentation on exports (autodoc_errors) 143
error_text/3 (pred) 143

28 Resolution of Bibliographical References 145
28.1 Usage and interface (autodoc_bibrefs) 145
28.2 Documentation on exports (autodoc_bibrefs) 145
resolve_bibliography/1 (pred) 145

29 Auxiliary Definitions 147
29.1 Usage and interface (autodoc_aux) 147
29.2 Documentation on exports (autodoc_aux) 147
read_file/2 (pred) 147
ascii_blank_lines/2 (pred) 147
sh_exec/2 (pred) 147

30 Image Handling 149
30.1 Usage and interface (autodoc_images) 149
30.2 Documentation on exports (autodoc_images) 149
locate_and_convert_image/4 (pred) 149

References 151

Library/Module Definition Index 153

Predicate/Method Definition Index 155

Property Definition Index 157

Regular Type Definition Index 159

Declaration Definition Index 161

Concept Definition Index **163**

Author Index **165**

Global Index **167**

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 several Un*x-related utilities. It is possible to run lpdoc under Win32 using Cygwin.

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 14 [Installing lpdoc], page 101.

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 12). 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 15).

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 12).

If the manual is generated from a single main file (i.e., `component/1`, 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 12).

Additionally, `lpdoc` can provide some data from the main (prolog) documentation file. For this purpose the option `getinfo` can be used instead of specifying the format. This option reads the asked fields from `getinfo` variable (defined in `SETTINGS.pl` or via arguments with -d option). `lpdoc` will generate files with main documentation file-name as base-name, followed by one underscore, the asked field (got from `getinfo`), and the extension. The content of each of these files (so also the extension) is specified by `getinfo_format`, that can take the values `html, ascii, texic`. For example, to ask for the `summary` and the `author` fields from a prolog file called `file.pl`, with `lpdoc` documentation, we can execute the command `lpdoc -d getinfo=[author,summary] getinfo`. The files `myfile_author.txt` and `myfile_summary.txt` will be created. If also the option `-d getinfo_format=html` is used, the files will have html extension (and content).

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 an `SETTINGS` file (see Section 2.2 [Generating a manual], page 9).

> TODO: command line options not available here; need cooperation with lpmake

PART I - LPdoc Reference Manual

2 Generating Installing and Accessing Manuals

Author(s): Manuel Hermenegildo.

Note: significant parts of this are obsolete. They must be updated to describe lpdoc version 2.0.

This section 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.pl* 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

Two possible scenarios are described in this section. The first one is indicated to document quickly a single module and the second one targets the documentation of a larger application or library, in which the settings (which define how the documentation is to be generated, etc.) are read from a file, so that they can be reused as the application / library evolves.

In order to make `lpdoc` generate quickly the documentation of a single file it suffices to execute the command `lpdoc -d doc_structure=modulename dvi`, where `modulename` is the module to be documented (without extension) and (in this example) `dvi` is the desired format of the manual (other accepted formats include html, pfd, ps, etc. – see later). `lpdoc` will generate a manual with the name of the module and the format extension (in the example it would be modulename.dvi) in the same directory where it is executed.

For the second scenario, the `lpdoc` library directory includes a generic file which is quite useful for the generation of complete manuals: the *SETTINGS.pl* file. Use of this file is strongly recommended. Generating a manual using this file involves the following steps:

- Create a directory (e.g., `doc`) in which the documentation will be built. The creation of this directory is recommended, as it will be populated with intermediate files which are best kept separate. This directory is typically created in the top directory of the distribution of the application or library to be documented.
- Execute the command `lpdoc lpsettings` in the directory where the documentation is to be created (e.g., `doc` in the previous point). `lpdoc` will create an `SETTINGS.pl.generated` file with the default settings. This file should be renamed to `SETTINGS.pl` once the user agrees with its contents.
- Edit `SETTINGS.pl` to suit your needs. It is recommended that you review, at least, the following points:

- Set the variable `filepath` to include all the directories where the files to be documented can be found.
- Set the variable `systempath` to include all the *system* directories where system files used can be found, regardless whether they are to be documented or not. This will be used to access definitions of types, etc.

  It is very important to include *all* related directories either in `filepath` or in `systempath` 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 `doc_structure` to be the *document structure* (`doc_structure/1`).

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

- `doc_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 `autodoc` or type `lpdoc -help` for a list of these options.
- In the same way `doc_compopts` sets options for the component files. Currently these options are common to all component files but they can be different from `doc_mainopts`. The allowable options are the same as above.
- `docformat` determines the set of formats (`dvi`, `ps`, `ascii`, `html`, `info`, `manl`, ...) in which the documentation should be generated by default when typing `lpdoc 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*).
- `output_name` determines the base file name of the main documents generated by lpdoc. By default it is equal to the main file name, or, if the main file name ends with `_doc`, then it is equal to the name without the `_doc` suffix. This is useful when the name of the documentation file to be produced needs to have a name that is not directly related to the main file being documented.
- `index` 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 `autodoc` 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.
- `bibfile` determines a list of *.bib files* (one file per path), 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 `-nobiblio` 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. It 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. It does not save trees.

- Type `lpdoc all` to generate all the formats defined. `lpdoc dvi`, `lpdoc html`, `lpdoc ps` or `lpdoc info`, etc. will force the generation of a single target format.

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 `lpdoc dvi`). The resulting output can be easily viewed by tools such as `xdvi` (which can be started by simply typing `lpdoc view`). Note that once an `xdvi` window is started, it is not necessary to restart it every time the document is reformatted (`lpdoc 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

`lpdoc` can also take care of tidying up the directory where the documentation is being generated:

- `lpdoc 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.
- `lpdoc 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.
- `lpdoc docsclean` 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 14.2 [Other software packages required (lpdoc)], page 101), in order to generate the manuals in the target formats during installation.

- `lpdoc realclean` performs a complete cleanup, deleting also the .texic files, i.e., it typically leaves only the `SETTINGS.pl` file. This is is the most compact, but requires the presence of the tools mentioned above, the source files from which the manuals are generated and `lpdoc` in order to re generate the manuals in the target formats during installation.

2.5 Installing a generated manual in a public area

Note: This part is obsolete. It must be updated to describe lpdoc version 2.0. – EMM

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.pl` file. This is done by typing `lpdoc 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, `lpdoc 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 `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 `dir` files.Several manuals, coming from different `doc` directories, can be installed in the same `docdir` directory. In this case, the descriptions of and pointers to the different manuals will be automatically combined (appearing in alphabetic order) in the `index.html` and/or `dir` indices, and a *contents area* will appear at the beginning of the *html index page*. **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.Typing `lpdoc uninstall` in a `doc` directory will uninstall 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, `lpdoc uninstall` should be made before changing the `docformats` variable and doing `lpdoc 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 `doc/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.pl` file and typing `lpdoc 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., `Firefox` (a command `lpdoc htmlview` is available which, if there is an instance of a web browser 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.pl` file to this directory and typing `lpdoc 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.pl file to this directory and typing lpdoc 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.pl file to this directory, including the infoindex option in docformats, and typing lpdoc install. This will install the info manual in directory docdir and update the dir file there. lpdoc 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.pl file to this directory and typing lpdoc 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.pl` file, and selecting `infoindex` and `htmlindex` for the `docformats` variable, `lpdoc install`/`lpdoc 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 `lpdoc 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.pl` 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 (`:- doc(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 `doc/2` declaration (`:- doc(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 `:- doc(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 `:- doc(filetype,part).` should be inserted in the sequence of files that make up the `components` variable of the `SETTINGS.pl` file at each point in which a major part starts. The `:- doc(title,"...").` declaration of this file will be used as the part title, and the `:- doc(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 `doc/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.pl` 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 `doc/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:

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

will include the contents of the file `Intro.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 in the `lpdoc` `SETTINGS.pl` 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)

Note: significant parts of this are obsolete. They must be updated to describe lpdoc version 2.0.

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 `INSTALLATION.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:

```
:- doc(module,"
   ...
   @section{Installation instructions}
   @include{INSTALLATION.lpdoc}
   ...
   ").
```

At the same time, this text can be used to generate a nicely formatted INSTALLATION 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]).
:- doc(title,"Installation instructions").
:- doc(module,"@include{INSTALLATION.lpdoc}").
main. %% forces file to be documented as an application
```

Then, the ascii INSTALLATION file can be generated by simply running lpdoc ascii in a directory with a SETTINGS.pl file where MAIN is set to INSTALLATION.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.pl 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.pl file than the maximum number supported by the installed version of tex/ texinfo installations, as mentioned in Section 2.2 [Generating a manual], page 9. 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 LPdoc Comments and Mark-up Language

Author(s): Manuel Hermenegildo.

This defines the admissible uses of the `doc/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 `doc/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:**
  - *Predicates:*

    `doc_id_type/3`.
  - *Properties:*

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

    `version_descriptor/1`, `filetype/1`.
- **Other modules used:**
  - *System library modules:*

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

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

3.2 Documentation on exports (`comments`)

docstring/1: PROPERTY

Defines the format of the character strings which can be used in machine readable comments (`doc/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*.

`@author{`*text*`}`
: *text* will be printed (in a normal font). This command is used to reference the name of an author (not necessarily establishing the module authorship).

- **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.

`@begin{alert}`
: marks the beginning of a section of text in a *framed box*, for alert messages.

`@end{alert}`
: marks the end of the alert message.

@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.

- **Mathematics:**

  The following commands are used to format text in mathematical .

  @math{*text*}
  : in-line typeset the *text* formula.

  `@begin{displaymath}`
  : marks the beginning of a formula (useful for long formulas).

`@end{displaymath}`
marks the end of the (long) formula.

`@defmathcmd{`*cmd*`}{`*n*`}{`*def*`}`
defines the math command *cmd*, taking *n* arguments, which is expanded as *def*. Arguments are denotated as `#1`, ..., `#n` inside *def*.

`@defmathcmd{`*cmd*`}{`*def*`}`
defines the math command *cmd*, which is expanded as *def* (with no arguments).

- **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}`	⇒ 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. Note that this only means that the file will be included as is. If you want the string to be represented in verbatim mode in the output, you must surround the `@includeverbatim{`*filename*`}` with `@begin{verbatim}` and `@end{verbatim}`.

  `@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_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.

doc_id_type/3: PREDICATE

No further documentation available for this predicate.

3.3 Documentation on internals (`comments`)

doc/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: :- `doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=title` (`=` /2)

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

Usage 2: :- `doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=subtitle` (`=/2`)

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

Usage 3: `:- doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=author` (`=/2`)

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

Usage 4: `:- doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=ack` (`=/2`)

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

Usage 5: `:- doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=copyright` (`=/2`)

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

Usage 6: `:- doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=summary` (`=/2`)

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

Usage 7: :- `doc(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 **sections** if substructure is needed.
- *Example:*

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

- *The following properties should hold upon exit:*

  `CommentType=module` (`=/2`)

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

Usage 8: :- `doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=appendix` (`=/2`)

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

Usage 9: :- `doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=usage` (`=/2`)

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

Usage 10: :- `doc(CommentType,Section)`.

- *Description:* Insert a *program section* with name `Section`. Sectioning commands allow a structured separation of the program into parts. The division is only for documentation purposes, so visibility and scope of definitions is not affected by sectioning commands.
- *Example:*

```
:- doc(section,"Main Steps of the Algorithm").
```

- *The following properties should hold upon exit:*

  `CommentType=section` (`=/2`)

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

Usage 11: :- `doc(CommentType,SubSection)`.

- *Description:* Insert a *program subsection* with name `SubSection` (see *program section* command for more details).
- *Example:*

```
:- doc(subsection,"Auxiliary Definitions").
```

- *The following properties should hold upon exit:*

  `CommentType=subsection` (`=` /2)

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

Usage 12: :- `doc(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:*

```
:- doc(doc/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:

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

  (`predname/1`)

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

Usage 13: :- `doc(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:*

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

- *The following properties should hold upon exit:*

  `CommentType=bug` (`=` /2)

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

Usage 14: :- `doc(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:*

```
:- doc(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 15: :- `doc(CommentType,VersionMaintenanceType)`.

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

  ```
  :- doc(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 `doc/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=version_maintenance` (`= /2`)

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

Usage 16: :- `doc(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 `:- doc(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 `:- doc(doinclude,PredName).` will force the documenation for `PredName` to appear directly in this module.
- *Example:*

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

  `CommentType=doinclude` (`= /2`)

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

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

  (`predname/1`)

Usage 17: :- `doc(CommentType,PredName)`.

- *Description:* A different usage which allows the second argument of `:- doc(doinclude,...)` to be a list of predicate names.
- *The following properties should hold upon exit:*

  `CommentType=doinclude` (`=/2`)

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

Usage 18: :- `doc(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:*

  ```
  :- doc(hide,p/3).
  ```

- *The following properties should hold upon exit:*

  `CommentType=hide` (`=/2`)

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

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

  (`predname/1`)

Usage 19: :- `doc(CommentType,PredName).`

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

  `CommentType=hide` (`=/2`)

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

Usage 20: :- `doc(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 `doc/2` declaration of this kind strictly only needs to be added to user-type files.

  Example:

  ```
  :- doc(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 16 for details.
- *The following properties should hold upon exit:*

  `CommentType=filetype` (`=/2`)

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

Usage 21: :- `doc(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:*

  ```
  :- doc(nodoc,assertions).
  ```

- *The following properties should hold upon exit:*

  `CommentType=nodoc` (`=`/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 `doc(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.

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 contain formal specifications (such as sets of preconditions, post-conditions, or descriptions of computations) as well as 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 [PBH00]. 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_doc)

- **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]).
  :- 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], test/1 [1150,fx], test/2 [1150,xfx], texec/1 [1150,fx], texec/2 [1150,xfx], comp/1 [1150,fx], comp/2 [1150,xfx], entry/1 [1150,fx], exit/1 [1150,fx], exit/2 [1150,xfx].
- **New declarations defined:**

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

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

    term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

4.4 Documentation on new declarations (assertions_doc)

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: :- `AssertionStatus` `pred` `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`)

texec/1: DECLARATION

This assertion is similar to a `calls/1` assertion but it is used to provide input data and execution commands to the unit-test driver.

Usage: :- `texec` `AssertionBody`.

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

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

texec/2: DECLARATION

This assertion is similar to a `texec/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `texec/1` assertions are assumed to have `check` status.

Usage: :- `AssertionStatus` `texec` `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`)

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 with an assertion status. Non-qualified `calls/1` assertions are assumed to have `check` status.

Usage: :- `AssertionStatus calls 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

success assertion This assertion is similar to a `success/1` assertion but it is explicitely qualified with an assertion status. The status of non-qualified `success/1` assertions is assumed to be `check`.

Usage: :- `AssertionStatus success 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`)

test/1: DECLARATION

This assertion is similar to a success assertion but it specifies a concrete test case to be run in order verify (partially) that the predicate is working as expected. For example, the following test will verify that the length predicate works well for the particular list given:

```
    :- test length(L,N) : ( L = [1,2,5,2] ) => ( N = 4 ).
```

Usage: :- `test AssertionBody`.

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

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

test/2: DECLARATION

This assertion is similar to a `test/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `test/1` assertions are assumed to have `check` status. In this context, check means that the test should be executed when the developer runs the test battery.

Usage: :- `AssertionStatus test 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: :- `AssertionStatus comp 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 6 [Declaring regular types], page 51 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: :- `AssertionStatus prop 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`)

exit/1: DECLARATION

This type of assertion provides information about the answers that an (exported) predicate provides for *external* calls. It is identical syntactically to a `success/1` assertion. However, it describes only external answers, i.e., answers to the exported predicates of a module from outside the module, or answers to the predicates in a non-modular file from other files (or the user). The described answers may be conditioned to a particular way of calling the predicate. E.g.:

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

Usage: :- `exit AssertionBody`.

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

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

exit/2: DECLARATION

exit assertion This assertion is similar to an `exit/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `exit/1` assertions are assumed the qualifier `check`.

Usage: :- `AssertionStatus exit 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`)

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: :- `AssertionStatus decl 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`)

doc/2: DECLARATION

Usage: :- `doc(Pred,Comment)`.

- *Description:* Documentation . This assertion provides 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`)

comment/2: DECLARATION

Usage: :- `comment(Pred,Comment)`.

- *Description:* An alias for `doc/2` (deprecated, for compatibility with older versions).
- *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_doc`)

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 Chapter 12 [Run-time checking of assertions], page 97.
- *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.

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

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 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 variable, 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,+Parameter,Output) is valid, as long as +/1 is declared as a mode.

  Acceptable modes are documented in library(basicmodes) and library(isomodes). User defined modes are documented in modedef/1.
- Any term. In this case this term determines the instantiation state of the corresponding argument position of the predicate calls to which the assertion applies.
- A ground term preceded by a “ mode.” The ground term determines a property of the corresponding argument. The mode determines if it applies to the calls and/or the successes. 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. For example, the head pattern p(Input,+list(int),Output) is valid for mode +/1 defined in library(isomodes), and equivalent in this case to having the head pattern p(Input,A,Output) and stating that the property list(A,int) holds for the calls of the predicate.
- Any term preceded by a “ mode.” In this case, only one variable is admitted, it has to be the first argument of the mode, and it represents the argument position. I.e., it plays the role of the new variable mentioned above. Thus, no rewriting of the head pattern is performed in this case. For example, the head pattern p(Input,+(Parameter,list(int)),Output) is valid for mode +/2 defined in library(isomodes), and equivalent in this case to having the head pattern p(Input,Parameter,Output) and stating that the property list(Parameter,int) holds for the calls of the predicate.

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(exit).
assrt_type(test).
assrt_type(texec).
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 Declaring regular types

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

This library package adds declarations and new operator definitions which provide simple syntactic sugar to write regular type definitions in source code. Regular types are just properties which have the additional characteristic of being regular types (`basic_props:regtype/1`), defined below.

For example, this library package allows writing:

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

instead of the more cumbersome:

```
:- 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.

6.1 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 Chapter 12 [Run-time checking of assertions], page 97) 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 runtime 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.

6.2 Usage and interface (`regtypes_doc`)

- **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:*

    `term_basic`.

6.3 Documentation on new declarations (`regtypes_doc`)

regtype/1: DECLARATION

This assertion is similar to a prop assertion but it flags that the property being documented is also a " regular type." Regular types are properties whose definitions are *regular programs* (see lelow). This allows for example checking whether it is in the class of types supported by the regular type checking and inference modules.

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.

A parametric type functor is a regular type, defined by a regular program, or a basic type. Basic types are defined in Chapter 7 [Basic data types and properties], page 55.

The set of regular 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: :- `AssertionStatus regtype 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`)

7 Basic data types and properties

Author(s): Daniel Cabeza, Manuel Hermenegildo.

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.

7.1 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`, `inst/2`, `iso/1`, `deprecated/1`, `not_further_inst/2`, `sideff/2`, `regtype/1`, `native/1`, `native/2`, `no_rtcheck/1`, `eval/1`, `equiv/2`, `bind_ins/1`, `error_free/1`, `memo/1`, `filter/2`, `pe_type/1`.
  - *Regular Types:*
    `term/1`, `int/1`, `nnegint/1`, `flt/1`, `num/1`, `atm/1`, `struct/1`, `gnd/1`, `gndstr/1`, `constant/1`, `callable/1`, `operator_specifier/1`, `list/1`, `list/2`, `nlist/2`, `sequence/2`, `sequence_or_list/2`, `character_code/1`, `string/1`, `num_code/1`, `predname/1`, `atm_or_atm_list/1`, `flag_values/1`.
- **Other modules used:**
  - *System library modules:*
    `assertions/native_props`, `terms_check`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

7.2 Documentation on exports (`basic_props`)

term/1: REGTYPE

The most general type (includes all possible terms).

General properties: `term(X)`

- *The following properties hold globally:*
  `term(X)` is side-effect `free`. (`sideff/2`)

`term(X)`

- *The following properties hold globally:*
  `term(X)` is evaluable at compile-time. (`eval/1`)

`term(X)`

- *The following properties hold globally:*
  `term(X)` is equivalent to `true`. (`equiv/2`)

Usage: `term(X)`

- *Description:* `X` is any term.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

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.

General properties: `int(T)`

- *The following properties hold globally:*
  `int(T)` is side-effect `free`. (`sideff/2`)

`int(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `int(T)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `int(T)` are deterministic. (`is_det/1`)

`int(T)`

- *The following properties hold upon exit:*
  `T` is an integer. (`int/1`)
- *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `int(T)`

- *Description:* `T` is an integer.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

nnegint/1: REGTYPE

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

General properties: `nnegint(T)`

- *The following properties hold globally:*
  `nnegint(T)` is side-effect `free`. (`sideff/2`)

`nnegint(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `nnegint(T)` is evaluable at compile-time. (`eval/1`)

`nnegint(T)`

- *The following properties hold upon exit:*
  `T` is a non-negative integer. (`nnegint/1`)

- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `nnegint(T)`

- *Description:* `T` is a non-negative integer.
- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`native/1`)

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`

General properties: `flt(T)`

- *The following properties hold globally:*

  `flt(T)` is side-effect `free`. (`sideff/2`)

`flt(T)`

- *If the following properties hold at call time:*

  `T` is currently a term which is not a free variable. (`nonvar/1`)

  then the following properties hold globally:

  `flt(T)` is evaluable at compile-time. (`eval/1`)

  All calls of the form `flt(T)` are deterministic. (`is_det/1`)

`flt(T)`

- *The following properties hold upon exit:*

  `T` is a float. (`flt/1`)
- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `flt(T)`

- *Description:* `T` is a float.
- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`native/1`)

num/1: REGTYPE

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

General properties: `num(T)`

- *The following properties hold globally:*

  `num(T)` is side-effect `free`. (`sideff/2`)

  `num(T)` is binding insensitive. (`bind_ins/1`)

`num(T)`

- *If the following properties hold at call time:*

  `T` is currently a term which is not a free variable. (`nonvar/1`)

  then the following properties hold globally:

  `num(T)` is evaluable at compile-time. (`eval/1`)

  All calls of the form `num(T)` are deterministic. (`is_det/1`)

`num(T)`

- *The following properties hold upon exit:*

  `T` is a number. (`num/1`)
- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `num(T)`

- *Description:* `T` is a number.
- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`native/1`)

atm/1: REGTYPE

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

General properties: `atm(T)`

- *The following properties hold globally:*

  `atm(T)` is side-effect `free`. (`sideff/2`)

`atm(T)`

- *If the following properties hold at call time:*

  `T` is currently a term which is not a free variable. (`nonvar/1`)

  then the following properties hold globally:

  `atm(T)` is evaluable at compile-time. (`eval/1`)

  All calls of the form `atm(T)` are deterministic. (`is_det/1`)

`atm(T)`

- *The following properties hold upon exit:*

  `T` is an atom. (`atm/1`)
- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `atm(T)`

- *Description:* `T` is an atom.
- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`native/1`)

struct/1: REGTYPE

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

General properties: `struct(T)`

- *The following properties hold globally:*
  `struct(T)` is side-effect `free`. (`sideff/2`)

`struct(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `struct(T)` is evaluable at compile-time. (`eval/1`)

`struct(T)`

- *The following properties hold upon exit:*
  `T` is a compound term. (`struct/1`)

Usage: `struct(T)`

- *Description:* `T` is a compound term.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

gnd/1: REGTYPE

The type of all terms without variables.

General properties: `gnd(T)`

- *The following properties hold globally:*
  `gnd(T)` is side-effect `free`. (`sideff/2`)

`gnd(T)`

- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `gnd(T)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `gnd(T)` are deterministic. (`is_det/1`)

`gnd(T)`

- *The following properties hold upon exit:*
  `T` is ground. (`gnd/1`)
- *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`test_type/2`)

Usage: `gnd(T)`

- *Description:* `T` is ground.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

gndstr/1: REGTYPE

General properties: `gndstr(T)`

- *The following properties hold globally:*
  `gndstr(T)` is side-effect `free`. (`sideff/2`)

`gndstr(T)`

- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `gndstr(T)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `gndstr(T)` are deterministic. (`is_det/1`)

`gndstr(T)`

- *The following properties hold upon exit:*
  `T` is a ground compound term. (`gndstr/1`)

Usage: `gndstr(T)`

- *Description:* `T` is a ground compound term.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

constant/1: REGTYPE

General properties: `constant(T)`

- *The following properties hold globally:*
  `constant(T)` is side-effect `free`. (`sideff/2`)

`constant(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `constant(T)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `constant(T)` are deterministic. (`is_det/1`)

`constant(T)`

- *The following properties hold upon exit:*
  `T` is an atomic term (an atom or a number). (`constant/1`)

Usage: `constant(T)`

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

callable/1: REGTYPE

General properties: `callable(T)`

- *The following properties hold globally:*
  `callable(T)` is side-effect `free`. (`sideff/2`)

`callable(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `callable(T)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `callable(T)` are deterministic. (`is_det/1`)

`callable(T)`

- *The following properties hold upon exit:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)

Usage: `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.

General properties: `operator_specifier(X)`

- *The following properties hold globally:*
  `operator_specifier(X)` is side-effect `free`. (`sideff/2`)

`operator_specifier(X)`

- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `operator_specifier(X)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `operator_specifier(X)` are deterministic. (`is_det/1`)
  Goal `operator_specifier(X)` produces `7` solutions. (`relations/2`)

`operator_specifier(T)`

- *The following properties hold upon exit:*
  `T` specifies the type and associativity of an operator. (`operator_specifier/1`)

Usage: `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).
```

General properties: `list(L)`

- *The following properties hold globally:*
  `list(L)` is side-effect `free`. (`sideff/2`)

`list(L)`

- *If the following properties hold at call time:*
  `L` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `list(L)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `list(L)` are deterministic. (`is_det/1`)

`list(T)`

- *The following properties hold upon exit:*
  `T` is a list. (`list/1`)

Usage: `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))`.

General properties: `list(L,T)`

- *The following properties hold globally:*
  `list(L,T)` is side-effect `free`. (`sideff/2`)

`list(L,T)`

- *If the following properties hold at call time:*
  `L` is currently ground (it contains no variables). (`ground/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `list(L,T)` is evaluable at compile-time. (`eval/1`)

`list(X,T)`

- *The following properties hold upon exit:*
  `X` is a list. (`list/1`)

Usage: `list(L,T)`

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

nlist/2: REGTYPE

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

General properties: `nlist(L,T)`

- *The following properties hold globally:*
  `nlist(L,T)` is side-effect `free`. (`sideff/2`)

`nlist(L,T)`

- *If the following properties hold at call time:*
  `L` is currently ground (it contains no variables). (`ground/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `nlist(L,T)` is evaluable at compile-time. (`eval/1`)

`nlist(X,T)`

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

Usage: `nlist(L,T)`

- *Description:* `L` is `T` or a nested list of `T`s. Note that if `T` is term, this type is equivalent to term, this fact explain why we do not have a `nlist/1` type

member/2: PROPERTY

General properties: `member(X,L)`

- *The following properties hold globally:*
  `member(X,L)` is side-effect `free`. (`sideff/2`)
  `member(X,L)` is binding insensitive. (`bind_ins/1`)

`member(X,L)`

- *If the following properties hold at call time:*
  `L` is a list. (`list/1`)
  then the following properties hold globally:
  `member(X,L)` is evaluable at compile-time. (`eval/1`)

`member(_X,L)`

- *The following properties hold upon exit:*
  `L` is a list. (`list/1`)

`member(X,L)`

- *If the following properties hold at call time:*
  `L` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold upon exit:
  `X` is currently ground (it contains no variables). (`ground/1`)

Usage: `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))`.

General properties: `sequence(S,T)`

- *The following properties hold globally:*
  `sequence(S,T)` is side-effect `free`. (`sideff/2`)

`sequence(S,T)`
- *If the following properties hold at call time:*
  `S` is currently ground (it contains no variables). (`ground/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `sequence(S,T)` is evaluable at compile-time. (`eval/1`)

`sequence(E,T)`
- *The following properties hold upon exit:*
  `E` is currently a term which is not a free variable. (`nonvar/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)

Usage: `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))`.

General properties: `sequence_or_list(S,T)`
- *The following properties hold globally:*
  `sequence_or_list(S,T)` is side-effect `free`. (`sideff/2`)

`sequence_or_list(S,T)`
- *If the following properties hold at call time:*
  `S` is currently ground (it contains no variables). (`ground/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `sequence_or_list(S,T)` is evaluable at compile-time. (`eval/1`)

`sequence_or_list(E,T)`
- *The following properties hold upon exit:*
  `E` is currently a term which is not a free variable. (`nonvar/1`)
  `T` is currently ground (it contains no variables). (`ground/1`)

Usage: `sequence_or_list(S,T)`
- *Description:* `S` is a sequence or list of `T`s.

character_code/1: REGTYPE

General properties: `character_code(T)`
- *The following properties hold globally:*
  `character_code(T)` is side-effect `free`. (`sideff/2`)

`character_code(T)`
- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`nonvar/1`)
  then the following properties hold globally:
  `character_code(T)` is evaluable at compile-time. (`eval/1`)

`character_code(I)`

- *The following properties hold upon exit:*
  `I` is an integer which is a character code. (`character_code/1`)

Usage: `character_code(T)`

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

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]`.

General properties: `string(T)`

- *The following properties hold globally:*
  `string(T)` is side-effect `free`. (`sideff/2`)

`string(T)`

- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `string(T)` is evaluable at compile-time. (`eval/1`)

`string(T)`

- *The following properties hold upon exit:*
  `T` is a string (a list of character codes). (`string/1`)

Usage: `string(T)`

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

num_code/1: REGTYPE

These are the ASCII codes which can appear in decimal representation of floating point and integer numbers, including scientific notation and fractionary part.

predname/1: REGTYPE

General properties: `predname(P)`

- *The following properties hold globally:*
  `predname(P)` is side-effect `free`. (`sideff/2`)

`predname(P)`

- *If the following properties hold at call time:*
  `P` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `predname(P)` is evaluable at compile-time. (`eval/1`)

`predname(P)`

- *The following properties hold upon exit:*
  `P` is a Name/Arity structure denoting a predicate name:

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

(predname/1)

Usage: `predname(P)`

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

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

atm_or_atm_list/1: REGTYPE

General properties: `atm_or_atm_list(T)`

- *The following properties hold globally:*

  `atm_or_atm_list(T)` is side-effect `free`. (`sideff/2`)

`atm_or_atm_list(T)`

- *If the following properties hold at call time:*

  `T` is currently ground (it contains no variables). (`ground/1`)

  then the following properties hold globally:

  `atm_or_atm_list(T)` is evaluable at compile-time. (`eval/1`)

`atm_or_atm_list(T)`

- *The following properties hold upon exit:*

  `T` is an atom or a list of atoms. (`atm_or_atm_list/1`)

Usage: `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))`.

General properties: `compat(Term,Prop)`

- *If the following properties hold at call time:*

  `Term` is currently ground (it contains no variables). (`ground/1`)

  `Prop` is currently ground (it contains no variables). (`ground/1`)

  then the following properties hold globally:

  `compat(Term,Prop)` is evaluable at compile-time. (`eval/1`)

Usage: `compat(Term,Prop)`

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

inst/2: PROPERTY

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

General properties: `inst(Term,Prop)`

- *The following properties hold globally:*
  `inst(Term,Prop)` is side-effect `free`. (`sideff/2`)

`inst(Term,Prop)`

- *If the following properties hold at call time:*
  `Term` is currently ground (it contains no variables). (`ground/1`)
  `Prop` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `inst(Term,Prop)` is evaluable at compile-time. (`eval/1`)

Usage: `inst(Term,Prop)`

- *Description:* `Term` is instantiated enough to satisfy `Prop`.

iso/1: PROPERTY

Meta-predicate with arguments: `iso(goal)`.

General properties: `iso(G)`

- *The following properties hold globally:*
  `iso(G)` is side-effect `free`. (`sideff/2`)

Usage: `iso(G)`

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

deprecated/1: PROPERTY

Specifies that the predicate marked with this global property has been deprecated, i.e., its use is not recommended any more since it will be deleted at a future date. Typically this is done because its functionality has been superseded by another predicate.

Meta-predicate with arguments: `deprecated(goal)`.

General properties: `deprecated(G)`

- *The following properties hold globally:*
  `deprecated(G)` is side-effect `free`. (`sideff/2`)

Usage: `deprecated(G)`

- *Description:* **DEPRECATED.**

not_further_inst/2: PROPERTY

Meta-predicate with arguments: `not_further_inst(goal,?)`.

General properties: `not_further_inst(G,V)`

- *The following properties hold globally:*
  `not_further_inst(G,V)` is side-effect `free`. (`sideff/2`)

Usage: `not_further_inst(G,V)`

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

sideff/2: PROPERTY

`sideff(G,X)`

Declares that `G` is side-effect free (if its execution has no observable result other than its success, its failure, or its abortion), soft (if its execution may have other observable results which, however, do not affect subsequent execution, e.g., input/output), or hard (e.g., assert/retract).

Meta-predicate with arguments: `sideff(goal,?)`.

General properties: `sideff(G,X)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`native/1`)

  `sideff(G,X)` is side-effect `free`. (`sideff/2`)

Usage: `sideff(G,X)`

- *Description:* `G` is side-effect `X`.
- *If the following properties hold at call time:*

  `X` is an element of `[free,soft,hard]`. (`member/2`)

regtype/1: PROPERTY

Meta-predicate with arguments: `regtype goal`.

General properties: `regtype G`

- *The following properties hold globally:*

  `regtype G` is side-effect `free`. (`sideff/2`)

Usage: `regtype G`

- *Description:* Defines a regular type.

native/1: PROPERTY

Meta-predicate with arguments: `native(goal)`.

General properties: `native(P)`

- *The following properties hold globally:*

  `native(P)` is side-effect `free`. (`sideff/2`)

Usage: `native(Pred)`

- *Description:* This predicate is understood natively by CiaoPP.

native/2: PROPERTY

Meta-predicate with arguments: `native(goal,?)`.

General properties: `native(P,K)`

- *The following properties hold globally:*

  `native(P,K)` is side-effect `free`. (`sideff/2`)

Usage: `native(Pred,Key)`

- *Description:* This predicate is understood natively by CiaoPP as `Key`.

no_rtcheck/1: PROPERTY

Meta-predicate with arguments: `no_rtcheck(goal)`.

General properties: `no_rtcheck(G)`

- *The following properties hold globally:*
  `no_rtcheck(G)` is side-effect `free`. (`sideff/2`)

Usage: `no_rtcheck(G)`

- *Description:* Declares that the assertion in which this comp property appears must not be checked at run-time.

eval/1: PROPERTY

Meta-predicate with arguments: `eval(goal)`.

Usage: `eval(Goal)`

- *Description:* `Goal` is evaluable at compile-time.

equiv/2: PROPERTY

Meta-predicate with arguments: `equiv(goal,goal)`.

Usage: `equiv(Goal1,Goal2)`

- *Description:* `Goal1` is equivalent to `Goal2`.

bind_ins/1: PROPERTY

Meta-predicate with arguments: `bind_ins(goal)`.

Usage: `bind_ins(Goal)`

- *Description:* `Goal` is binding insensitive.

error_free/1: PROPERTY

Meta-predicate with arguments: `error_free(goal)`.

Usage: `error_free(Goal)`

- *Description:* `Goal` is error free.

memo/1: PROPERTY

Meta-predicate with arguments: `memo(goal)`.

Usage: `memo(Goal)`

- *Description:* `Goal` should be memoized (not unfolded).

filter/2: PROPERTY

Usage: `filter(Vars,Goal)`

- *Description:* `Vars` should be filtered during global control).

flag_values/1: REGTYPE

Usage: `flag_values(X)`

- *Description:* Define the valid flag values

pe_type/1: PROPERTY

Meta-predicate with arguments: `pe_type(goal)`.

Usage: `pe_type(Goal)`

- *Description:* `Goal` will be filtered in partial evaluation time according to the PE types defined in the assertion.

8 Properties which are native to analyzers

Author(s): Francisco Bueno, Manuel Hermenegildo, Pedro López, Edison Mera.

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:*

    `clique/1`, `clique_1/1`, `constraint/1`, `covered/1`, `covered/2`, `exception/1`, `exception/2`, `fails/1`, `finite_solutions/1`, `have_choicepoints/1`, `indep/1`, `indep/2`, `is_det/1`, `linear/1`, `mshare/1`, `mut_exclusive/1`, `no_choicepoints/1`, `no_exception/1`, `no_exception/2`, `no_signal/1`, `no_signal/2`, `non_det/1`, `nonground/1`, `not_covered/1`, `not_fails/1`, `not_mut_exclusive/1`, `num_solutions/2`, `solutions/2`, `possibly_fails/1`, `possibly_nondet/1`, `relations/2`, `sideff_hard/1`, `sideff_pure/1`, `sideff_soft/1`, `signal/1`, `signal/2`, `signals/2`, `size/2`, `size/3`, `size_lb/2`, `size_o/2`, `size_ub/2`, `size_metric/3`, `size_metric/4`, `steps/2`, `steps_lb/2`, `steps_o/2`, `steps_ub/2`, `tau/1`, `terminates/1`, `test_type/2`, `throws/2`, `user_output/2`, `intervals/2`.
- **Other modules used:**
  - *System library modules:*

    `terms_check`, `terms_vars`, `sort`, `lists`, `streams`, `file_utils`, `system`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`, `internals`.

8.2 Documentation on exports (`native_props`)

clique/1: PROPERTY

`clique(X)`

`X` is a set of variables of interest, much the same as a sharing group but `X` represents all the sharing groups in the powerset of those variables. Similar to a sharing group, a clique is often translated to `ground/1`, `indep/1`, and `indep/2` properties.

Usage: `clique(X)`

- *Description:* The clique pattern is `X`.
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP as `clique(X)`. (`native/2`)

clique_1/1: PROPERTY

`clique_1(X)`

`X` is a set of variables of interest, much the same as a sharing group but `X` represents all the sharing groups in the powerset of those variables but disregarding the singletons. Similar to a sharing group, a clique_1 is often translated to `ground/1`, `indep/1`, and `indep/2` properties.

Usage: `clique_1(X)`

- *Description:* The 1-clique pattern is `X`.
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `clique_1(X)`. (`native/2`)

constraint/1: PROPERTY

`constraint(C)`

`C` contains a list of linear (in)equalities that relate variables and `int` values. For example, `[A < B + 4]` is a constraint while `[A < BC + 4]` or `[A = 3.4, B >= C]` are not.

Usage: `constraint(C)`

- *Description:* `C` is a list of linear equations
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

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.

covered/2: PROPERTY

`covered(X,Y)`

All variables occuring in `X` occur also in `Y`.

Usage: `covered(X,Y)`

- *Description:* `X` is covered by `Y`.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

exception/1: PROPERTY

Meta-predicate with arguments: `exception(goal)`.

Usage: `exception(Goal)`

- *Description:* Calls of the form `Goal` throw an exception.

exception/2: PROPERTY

Meta-predicate with arguments: `exception(goal,?)`.

Usage: `exception(Goal,E)`

- *Description:* Calls of the form `Goal` throw an exception that unifies with `E`.

fails/1: PROPERTY

`fails(X)`

Calls of the form `X` fail.

Meta-predicate with arguments: `fails(goal)`.

Usage: `fails(X)`

- *Description:* Calls of the form `X` fail.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

finite_solutions/1: PROPERTY

`finite_solutions(X)`

Calls of the form `X` produce a finite number of solutions [DLGH97].

Meta-predicate with arguments: `finite_solutions(goal)`.

Usage: `finite_solutions(X)`

- *Description:* All the calls of the form `X` have a finite number of solutions.

have_choicepoints/1: PROPERTY

Meta-predicate with arguments: `have_choicepoints(goal)`.

Usage: `have_choicepoints(X)`

- *Description:* A call to `X` creates choicepoints.

indep/1: PROPERTY

Usage: `indep(X)`

- *Description:* The variables in pairs in `X` are pairwise independent.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `indep(X)`. (`native/2`)

indep/2: PROPERTY

Usage: `indep(X,Y)`

- *Description:* `X` and `Y` do not have variables in common.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `indep([[X,Y]])`. (`native/2`)

is_det/1: PROPERTY

`is_det(X)`

All calls of the form `X` are deterministic, i.e., produce at most one solution, or do not terminate. In other words, if `X` succeeds, it can only succeed once. It can still leave choice points after its execution, but when backtracking into these, it can only fail or go into an infinite loop.

Meta-predicate with arguments: `is_det(goal)`.

Usage: `is_det(X)`

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

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.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

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`.
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `sharing(X)`. (`native/2`)

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.

Meta-predicate with arguments: `mut_exclusive(goal)`.

Usage: `mut_exclusive(X)`

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

no_choicepoints/1: PROPERTY

Meta-predicate with arguments: `no_choicepoints(goal)`.

Usage: `no_choicepoints(X)`

- *Description:* A call to `X` does not create choicepoints.

no_exception/1: PROPERTY

Meta-predicate with arguments: `no_exception(goal)`.

Usage: `no_exception(Goal)`

- *Description:* Calls of the form `Goal` do not throw any exception.

no_exception/2: PROPERTY

Meta-predicate with arguments: `no_exception(goal,?)`.

Usage: `no_exception(Goal,E)`

- *Description:* Calls of the form `Goal` do not throw exception `E`.

no_signal/1: PROPERTY

Meta-predicate with arguments: `no_signal(goal)`.

Usage: `no_signal(Goal)`

- *Description:* Calls of the form `Goal` do not send any signal.

no_signal/2: PROPERTY

Meta-predicate with arguments: `no_signal(goal,?)`.

Usage: `no_signal(Goal,E)`

- *Description:* Calls of the form `Goal` do not send the signal `E`.

non_det/1: PROPERTY

`non_det(X)`

All calls of the form `X` are non-deterministic, i.e., produce several solutions.

Meta-predicate with arguments: `non_det(goal)`.

Usage: `non_det(X)`

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

nonground/1: PROPERTY

Usage: `nonground(X)`

- *Description:* `X` is not ground.
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `not_ground(X)`. (`native/2`)

not_covered/1: PROPERTY

`not_covered(X)`

There is some call of the form `X` for which there is no clause whose test succeeds [DLGH97].

Usage: `not_covered(X)`

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

not_fails/1: PROPERTY

`not_fails(X)`

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

Meta-predicate with arguments: `not_fails(goal)`.

Usage: `not_fails(X)`

- *Description:* All the calls of the form `X` do not fail.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

not_mut_exclusive/1: PROPERTY

`not_mut_exclusive(X)`

For calls of the form `X` more than one clause may succeed. I.e., clauses are not disjoint for some call.

Meta-predicate with arguments: `not_mut_exclusive(goal)`.

Usage: `not_mut_exclusive(X)`

- *Description:* For some calls of the form `X` more than one clause may succeed.

num_solutions/2: PROPERTY

Usage 1: `num_solutions(X,N)`

- *Description:* All the calls of the form `X` have `N` solutions.
- *If the following properties should hold at call time:*
  `X` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)
  `N` is an integer. (`int/1`)

Usage 2: `num_solutions(Goal,Check)`

- *Description:* For a call to `Goal`, `Check(X)` succeeds, where `X` is the number of solutions.
- *If the following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)
  `Check` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)

solutions/2: PROPERTY

Usage: `solutions(Goal,Sols)`

- *Description:* Goal `Goal` produces the solutions listed in `Sols`.
- *If the following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)
  `Sols` is a list. (`list/1`)

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.

Meta-predicate with arguments: `possibly_fails(goal)`.

Usage: `possibly_fails(X)`

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

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`.

relations/2: PROPERTY

`relations(X,N)`

The goal `X` produces `N` solutions. In other words, `N` is the cardinality of the solution set of `X`.

Meta-predicate with arguments: `relations(goal,?)`.

Usage: `relations(X,N)`

- *Description:* Goal `X` produces `N` solutions.

sideff_hard/1: PROPERTY

Meta-predicate with arguments: `sideff_hard(goal)`.

Usage: `sideff_hard(X)`

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

sideff_pure/1: PROPERTY

Meta-predicate with arguments: `sideff_pure(goal)`.

Usage: `sideff_pure(X)`

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

sideff_soft/1: PROPERTY

Meta-predicate with arguments: `sideff_soft(goal)`.

Usage: `sideff_soft(X)`

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

signal/1: PROPERTY

Meta-predicate with arguments: `signal(goal)`.

Usage: `signal(Goal)`

- *Description:* Calls of the form `Goal` throw a signal.

signal/2: PROPERTY

Meta-predicate with arguments: `signal(goal,?)`.

Usage: `signal(Goal,E)`

- *Description:* A call to `Goal` sends a signal that unifies with `E`.

signals/2: PROPERTY

Meta-predicate with arguments: `signals(goal,?)`.

Usage: `signals(Goal,Es)`

- *Description:* Calls of the form `Goal` can generate only the signals that unify with the terms listed in `Es`.

size/2: PROPERTY

Usage: `size(X,Y)`

- *Description:* `Y` is the size of argument `X`, for any approximation.

size/3: PROPERTY

Usage: `size(A,X,Y)`

- *Description:* `Y` is the size of argument `X`, for the approximation `A`.

size_lb/2: PROPERTY

`size_lb(X,Y)`

The minimum size of the terms to which the argument `Y` is bound 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,LGHD96].

Usage: `size_lb(X,Y)`

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

size_o/2: PROPERTY

Usage: `size_o(X,Y)`

- *Description:* The size of argument `X` is in the order of `Y`.

size_ub/2: PROPERTY

`size_ub(X,Y)`

The maximum size of the terms to which the argument `Y` is bound 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,LGHD96].

Usage: `size_ub(X,Y)`

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

size_metric/3: PROPERTY

Meta-predicate with arguments: `size_metric(goal,?,?)`.

Usage: `size_metric(Head,Var,Metric)`

- *Description:* `Metric` is the metric of the variable `Var`, for any approximation.

size_metric/4: PROPERTY

Meta-predicate with arguments: `size_metric(goal,?,?,?)`.

Usage: `size_metric(Head,Approx,Var,Metric)`

- *Description:* `Metric` is the metric of the variable `Var`, for the approximation `Approx`. Currently, `Metric` can be: `int/1`, `size/1`, `length/1`, `depth/2`, and `void/1`.

steps/2: PROPERTY

`steps(X,Y)`

The time (in resolution steps) spent by any call of the form `X` is given by the expression `Y`

Meta-predicate with arguments: `steps(goal,?)`.

Usage: `steps(X,Y)`

- *Description:* `Y` is the cost (number of resolution steps) of any call of the form `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]

Meta-predicate with arguments: `steps_lb(goal,?)`.

Usage: `steps_lb(X,Y)`

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

steps_o/2: PROPERTY

Meta-predicate with arguments: `steps_o(goal,?)`.

Usage: `steps_o(X,Y)`

- *Description:* `Y` is the complexity order of 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].

Meta-predicate with arguments: `steps_ub(goal,?)`.

Usage: `steps_ub(X,Y)`

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

tau/1: PROPERTY

`tau(Types)`

`Types` contains a list with the type associations for each variable, in the form `V/[T1,..,TN]`.Note that tau is used in object-oriented programs only

Usage: `tau(TypeInfo)`

- *Description:* `Types` is a list of associations between variables and list of types
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

terminates/1: PROPERTY

`terminates(X)`

Calls of the form `X` always terminate [DLGH97].

Meta-predicate with arguments: `terminates(goal)`.

Usage: `terminates(X)`

- *Description:* All calls of the form `X` terminate.

test_type/2: PROPERTY

Meta-predicate with arguments: `test_type(goal,?)`.

Usage: `test_type(X,T)`

- *Description:* Indicates the type of test that a predicate performs. Required by the nonfailure analyisis.

throws/2: PROPERTY

Meta-predicate with arguments: `throws(goal,?)`.

Usage: `throws(Goal,Es)`

- *Description:* Calls of the form `Goal` can throw only the exceptions that unify with the terms listed in `Es`.

user_output/2: PROPERTY

Meta-predicate with arguments: `user_output(goal,?)`.

Usage: `user_output(Goal,S)`

- *Description:* Calls of the form `Goal` write `S` to standard output.

intervals/2: PROPERTY

Usage: `intervals(X,Y)`

- *Description:* Data size `X` belongs to some interval in the list of intervals `Y`. The list of intervals `Y` represents union of its elements

instance/2: PROPERTY

Usage: `instance(Term1,Term2)`

- *Description:* `Term1` is an instance of `Term2`.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

9 Meta-properties

Author(s): Francisco Bueno.

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:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

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: `A prop 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: `A regtype 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*.

Usage: `callme(A,B)`

- *The following properties should hold at call time:*
  `A` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)

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.

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,q/2,r/1, p/1, p/5, u/3,
           long/1, w/1, mytype/1, t/5, s/1, q/1],
          [assertions,basicmodes,fsyntax,regtypes,hiord,nativeprops]).

%% 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(library(lists), [length/2]).
%:- use_module(bar).
:- ensure_loaded(foo).

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

:- doc(author,"Anonymous Author 1").
:- doc(author,"Anonymous Author 2").

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

:- doc(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
:- doc(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
%% :- doc(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)) :-
%%         T(X),
%%         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 itself
%% is documented.
:- doc(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.

foo(_).

%% 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.

:- doc(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.".

q(_).

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

p(_).

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

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

s(_).

:- doc(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.".

:- doc(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.".

t(_, _, _, _, _).

%% Some other miscellaneous assertions:
```

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

u(_, _, _).

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

mytype(_).

w(_).

:- doc(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).".

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

p(_, _, _, _, _) :- _ is 1.

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

11 Auto Documenter Output for the Example Module

Author(s): Anonymous Author 1, Anonymous Author 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/2`, `r/1`, `p/1`, `p/5`, `u/3`, `w/1`, `mytype/1`, `t/5`, `s/1`, `q/1`.
  - *Properties:*
    `long/1`.
  - *Regular Types:*
    `bar/1`, `baz/1`, `aorb/1`, `tree_of/2`, `list_or_aorb/2`.
  - *Multifiles:*
    `p/3`.
- **Other modules used:**
  - *Files of module* `user`*:*
    `foo`.
  - *System library modules:*
    `assertions/native_props`, `engine/basic_props`, `lists`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

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).
```

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`)

p/1: PREDICATE

Usage:
- *The following properties should hold at call time:*
  `Arg1` is a free variable. (`var/1`)
- *The following properties should hold upon exit:*
  `Arg1` is a list. (`list/1`)

p/5: PREDICATE

Usage: `p(Arg1,Arg2,Arg3,Arg4,A)`
- *Call and exit should be compatible with:*
  `Arg1` is an integer. (`int/1`)
  `Arg3` is a list of `int`s. (`list/2`)
- *The following properties should hold at call time:*
  `Arg2` is currently ground (it contains no variables). (`ground/1`)
  `Arg4` is a free variable. (`var/1`)
  `A` is currently a term which is not a free variable. (`nonvar/1`)
- *The following properties should hold upon exit:*
  `Arg1` is ground. (`gnd/1`)
  `Arg2` is currently ground (it contains no variables). (`ground/1`)
- *The following properties should hold globally:*
  `Arg3` is not further instantiated. (`not_further_inst/2`)
  `1+length(A)` is a lower bound on the cost of any call of the form `p(Arg1,Arg2,Arg3,Arg4,A)`. (`steps_lb/2`)

u/3: PREDICATE

Usage 1:
- *The following properties should hold at call time:*
  `Arg1` is currently a term which is not a free variable. (`nonvar/1`)
  `Arg2` is a free variable. (`var/1`)
- *The following properties should hold upon exit:*
  `Arg3` is ground. (`gnd/1`)

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.

w/1: PREDICATE

Usage:

- *Calls should, and exit will be compatible with:*

  `Arg1` is a list of `mytype`s. (`list/2`)
- *The following properties should hold at call time:*

  `Arg1` is currently a term which is not a free variable. (`nonvar/1`)

mytype/1: PREDICATE

No further documentation available for this predicate.

t/5: PREDICATE

Usage: `t(A,B,C,D,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`)

  `E` is a list. (`list/1`)
- *The following properties should hold at call time:*

  `A` is currently a term which is not a free variable. (`nonvar/1`)

  `B` is a free variable. (`var/1`)

  `B` is rather long. (`long/1`)
- *The following properties should hold upon exit:*

  `E` is ground. (`gnd/1`)

  `C` is ground. (`gnd/1`)

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

  `D` is not further instantiated. (`not_further_inst/2`)

  All the calls of the form `t(A,B,C,D,E)` do not fail. (`not_fails/1`)

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`)

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.

list/1: REGTYPE

General properties: `list(L)`

- *The following properties hold globally:*
  `list(L)` is side-effect `free`. (`sideff/2`)

`list(L)`

- *If the following properties hold at call time:*
  `L` is currently ground (it contains no variables). (`ground/1`)
  then the following properties hold globally:
  `list(L)` is evaluable at compile-time. (`eval/1`)
  All calls of the form `list(L)` are deterministic. (`is_det/1`)

`list(T)`

- *The following properties hold upon exit:*
  `T` is a list. (`list/1`)

Usage: `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`)

list/2: REGTYPE

General properties: `list(L,T)`

- *The following properties hold globally:*

  `list(L,T)` is side-effect `free`. (`sideff/2`)

`list(L,T)`

- *If the following properties hold at call time:*

  `L` is currently ground (it contains no variables). (`ground/1`)

  `T` is currently ground (it contains no variables). (`ground/1`)

  then the following properties hold globally:

  `list(L,T)` is evaluable at compile-time. (`eval/1`)

`list(X,T)`

- *The following properties hold upon exit:*

  `X` is a list. (`list/1`)

Usage: `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 Run-time checking of assertions

Author(s): Edison Mera.

This package provides a complete implementation of run-time checks of predicate assertions. The program is instrumented to check such assertions at run time, and in case a property does not hold, the error is reported. Note that there is also an older package called rtchecks, by David Trallero. The advantage of this one is that it can be used independently of CiaoPP and also has updated functionality.

There are two main applications of run-time checks:

- To improve debugging of certain predicates, specifying some expected behavior that is checked at run-time with the assertions.
- To avoid manual implementation of run-time checks that should be done in some predicates, leaving the code clean and understandable.

The run-time checks can be configured using prolog flags. Below we itemize the valid prolog flags with its values and a brief explanation of the meaning:

- `rtchecks_level`
  - `exports`: Only use rtchecks for external calls of the exported predicates.
  - `inner` : Use also rtchecks for internal calls. Default.
- `rtchecks_trust`
  - `no` : Disable rtchecks for trust assertions.
  - `yes` : Enable rtchecks for trust assertions. Default.
- `rtchecks_entry`
  - `no` : Disable rtchecks for entry assertions.
  - `yes` : Enable rtchecks for entry assertions. Default.
- `rtchecks_exit`
  - `no` : Disable rtchecks for exit assertions.
  - `yes` : Enable rtchecks for exit assertions. Default.
- `rtchecks_test`
  - `no` : Disable rtchecks for test assertions. Default.
  - `yes` : Enable rtchecks for test assertions. Used for debugging purposes, but is better to use the unittest library.
- `rtchecks_inline`
  - `no` : Instrument rtchecks using call to library predicates present in `rtchecks_rt.pl`, `nativeprops.pl` and `basic_props.pl`. In this way, space is saved, but sacrifying performance due to usage of meta calls and external methods in the libraries. Default.
  - `yes` : Expand library predicates inline as far as possible. In this way, the code is faster, because its avoids metacalls and usage of external methods, but the final executable could be bigger.
- `rtchecks_asrloc` Controls the usage of locators for the assertions in the error messages. The locator says the file and lines that contains the assertion that had failed. Valid values are:
  - `no` : Disabled.
  - `yes` : Enabled. Default.
- `rtchecks_predloc` Controls the usage of locators for the predicate that caused the run-time check error. The locator says the first clause of the predicate that the violated assertion refers to.
  - `no` : Disabled.

    - `yes` : Enabled, Default.
- `rtchecks_callloc`
    - `no` : Do not show the stack of predicates that caused the failure
    - `predicate`: Show the stack of predicates that caused the failure. Instrument it in the predicate. Default.
    - `literal` : Show the stack of predicates that caused the failure. Instrument it in the literal. This mode provides more information, because reports also the literal in the body of the predicate.
- `rtchecks_namefmt`
    - `long` : Show the name of predicates, properties and the values of the variables
    - `short` : Only show the name of the predicate in a reduced format. Default.

12.1 Usage and interface (`rtchecks_doc`)

- **Library usage:**

  `:- use_package(rtchecks).`

  or

  `:- module(...,...,[rtchecks]).`
- **Other modules used:**
    - *Internal (engine) modules:*

      `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

13 Unit Testing Library

Author(s): Edison Mera.

This library provides an extension of the Ciao assertion language which allows writing *unit tests*. The central idea is to use the assertion language to provide specifications of test cases for a given predicate. The package also provides some special properties that are convenient when specifying unit tests and the required run-time libraries.

In general, a *test assertion* is written as follows:

```
:- test predicate(A1, A2, ..., An)
   :  <Precondition>
   => <Postcondition>
   +  <Global properties>
   #  <Comment>.
```

Where the fields of the test assertion have the usual meaning in Ciao assertions, i.e., they contain conjunctions of properties which must hold at certain points in the execution. Here we give a somewhat more operational ("test oriented"), reading to these fields: `predicate/n` is the predicate to be tested. `Precondition` is a goal that is called before the predicate being tested, and can be used to generate values of the input parameters. `Postcondition` is a goal that should succeed after `predicate/n` has been called. The idea appears to be simple, but note that due to the non-determinism of logic programs, the test engine needs to test all the solutions that can be tested up to given limits (for example, a maximum number of solutions, or a given time-out). `Properties` specifies some global properties that the predicate should meet, for example, `not_fails` means that the program does not fail, `exception(error(a,b))` means that the program should throw the exception `error(a,b)`, and so on. But there are some specific properties that only applies to testing specified in the module unittest_props.pl, for example `times(N)` specifies that the given test should be executed N times, `try_sols(N)` specifies that the first N solutions of the predicate `predicate/n` are tested. `Comment` is a string that document the test.

A convenient way to run these tests is by selecting options in the CiaoDbg menu within the development environment:

1. `Run tests in current module`: execute only the tests specified in the current module.
2. `Run tests in all related modules`: execute the tests specified in the module and in all the modules being used by this.
3. `Show untested predicates`: show the *exported* predicates that do not have any test assertion.

13.1 Additional notes

1. The test assertions allow performing *unit* testing, i.e., in Ciao, performing tests *at the predicate level.*
2. The tests currently can only be applied to exported predicates.
3. If you need to write tests for predicates that are spread over several modules, but work together, then it is best to create a separate module, and reexport to the predicates required to build the test. This allows performing *integration testing*, using the same syntax of the unit tests.
4. The Ciao system includes a good (and growing) number of unit tests. To run all the tests among the other standard tests within the CiaoDE run the following (at the top level of the source tree):

   ```
   ./ciaosetup runtests
   ```

13.2 Usage and interface (unittest_doc)

- **Library usage:**

  :- use_module(library(unittest)).
- **Other modules used:**
  - *Internal (engine) modules:*

    term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

14 Installing lpdoc

Author(s): Manuel Hermenegildo.

This documentation is outdated.

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.

14.1 Installing the source distribution (lpdoc)

- Before installing lpdoc, you may want to read Section 14.2 [Other software packages required (lpdoc)], page 101. 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 if needed edit the file LPDOCSETTINGS.pl in a text editor, but in general the default options works well (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 LPDOCSETTINGS.pl 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 LPDOCSETTINGS.pl.
- Type lpmake all. This should create the lpdoc executable and compile related libraries.
- Type lpmake install. This should 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.

14.2 Other software packages required (lpdoc)

The most basic functionality of lpdoc (generating manuals in .texi format, short manual 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

15 Documentation Generation Library

Author(s): Manuel Hermenegildo, Jose F. Morales.

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.

Note: This part is obsolete. – JFMC

The output format in which the documentation is generated is defined by the backend modules (`autodoc_texinfo`, `autodoc_html`, `autodoc_man`, etc.).

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).

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).

15.1 Usage and interface (autodoc)

- **Library usage:**

  `:- use_module(library(autodoc)).`
- **Exports:**
  - *Predicates:*

    backend_ignores_components/1, backend_alt_format/2, top_suffix/2, index_comment/2, option_comment/2, docstate_backend/2, docstate_currmod/2, docstate_set_currmod/3, docstate_opts/2, docstate_set_opts/3, docstate_set_customdic/3, docstate_inputfile/2, doc_message/2, doc_message/3, doc_opt/2, doc_customdic_lookup/3, doc_customdic_replace/4, doc_customdic_get/3, doc_customdic_maybe_get/3, doc_doing_mainmod/1, doc_modname/2, docstate_has_index/2, all_indices/2, get_doc/4, reset_output_dir_db/0, ensure_output_dir_prepared/2, get_autodoc_opts/3, autodoc_gen_doctree/5, infodir_base/2, autodoc_compute_grefs/3, autodoc_translate_doctree/3, autodoc_finish/1, autodoc_gen_alternative/2, version_patch/2, version_date/2, version_numstr/2.
  - *Properties:*

    supported_option/1.
  - *Regular Types:*

    backend_id/1, docstate/1, modtype/1.
  - *Multifiles:*

    autodoc_finish_hook/1, autodoc_gen_alternative_hook/2.
- **Other modules used:**
  - *Application modules:*

    lpdocsrc(src(autodoc_settings)), lpdocsrc(src(autodoc_filesystem)), lpdocsrc(src(autodoc_structure)), lpdocsrc(src(autodoc_doctree)), lpdocsrc(src(autodoc_refsdb)), lpdocsrc(src(autodoc_parse)), lpdocsrc(src(autodoc_index)), lpdocsrc(src(comments)), lpdocsrc(src(autodoc_html_resources)).
  - *System library modules:*

    ttyout, format, aggregates, read, make/make_rt, dict, compiler/compiler, assertions/assrt_lib, compiler/c_itf, assertions/assertions_props, messages, filenames, lists, terms, make/system_extra, system.
  - *Internal (engine) modules:*

    term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

15.2 Documentation on exports (autodoc)

backend_id/1: REGTYPE

Usage: `backend_id(Id)`

- *Description:* `Id` is a supported backend.

backend_ignores_components/1: PREDICATE

Usage: `backend_ignores_components(Id)`

- *Description:* `Id` does not take into account components (only documents the *mainfile*)

backend_alt_format/2: PREDICATE

Usage: `backend_alt_format(Id,Ext)`

- *Description:* `Ext` is an alternative file format that can be generated by the backend `Id`

top_suffix/2: PREDICATE

Usage: `top_suffix(FileFormat,PrincipalExt)`

- *Description:* `PrincipalExt` is extension of the target file that will generate the file with `FileFormat` extension.

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`)

supported_option/1: PROPERTY

Usage: `supported_option(Option)`

- *Description:* `Option` is a supported documentation option.

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 text. "
option_comment('-noundefined',"Do not signal undefined properties in text. "
option_comment('-nopropsepln',"Do not put each property in a separate line."
option_comment('-nobiblio',"Do not include a bibliographical 'References' ap
option_comment('-nosysmods',"Do not include system modules in list of
                                libraries used.").
option_comment('-noengmods',"Do not include system engine modules in list
                                of libraries used.").
option_comment('-noisoline',"Do not include *textual* description that a
                                given usage conforms to the ISO standard.").
option_comment('-propmods',"Include module name to which props belong.").
option_comment('-nopropuses',"Do not Include property uses (from assertions)
option_comment('-shorttoc',"Produce shorter table of contents (no entries
                                for individual defs of preds, props, etc.)."
option_comment('-regtypeprops',"Include in the doc for regtypes the global
                                prop stating that they are indeed regtypes."
option_comment('-onesided',"For printing on one side (default is two).").
option_comment('-nomath',"Disable mathematical environments.").
```

– *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`)

docstate/1: REGTYPE

A regular type, defined as follows:

```
docstate(docstate(Backend,Name,Opts,CustomDic,I)) :-
        backend_id(Backend),
        atom(Name),
        list(Opts,supported_option),
        dictionary(CustomDic),
        filename(I).
```

docstate_backend/2: PREDICATE

No further documentation available for this predicate.

docstate_currmod/2: PREDICATE

No further documentation available for this predicate.

docstate_set_currmod/3: PREDICATE

No further documentation available for this predicate.

docstate_opts/2: PREDICATE

No further documentation available for this predicate.

docstate_set_opts/3: PREDICATE

No further documentation available for this predicate.

docstate_set_customdic/3: PREDICATE

No further documentation available for this predicate.

docstate_inputfile/2: PREDICATE

No further documentation available for this predicate.

doc_message/2: PREDICATE

No further documentation available for this predicate.

doc_message/3: PREDICATE

No further documentation available for this predicate.

doc_opt/2: PREDICATE

No further documentation available for this predicate.

doc_customdic_lookup/3: PREDICATE

No further documentation available for this predicate.

doc_customdic_replace/4: PREDICATE

No further documentation available for this predicate.

doc_customdic_get/3: PREDICATE

No further documentation available for this predicate.

doc_customdic_maybe_get/3: PREDICATE

No further documentation available for this predicate.

doc_doing_mainmod/1: PREDICATE

No further documentation available for this predicate.

doc_modname/2: PREDICATE

Usage: `doc_modname(DocSt,ModName)`

– *Description:* `ModName` is the name of the module that we are documenting.

docstate_has_index/2: PREDICATE

No further documentation available for this predicate.

all_indices/2: PREDICATE

No further documentation available for this predicate.

get_doc/4: PREDICATE

No further documentation available for this predicate.

modtype/1: REGTYPE

```
modtype(part).
modtype(application).
modtype(module).
modtype(user).
modtype(include).
modtype(package).
```

Usage:

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

reset_output_dir_db/0: PREDICATE

No further documentation available for this predicate.

ensure_output_dir_prepared/2: PREDICATE

Usage: `ensure_output_dir_prepared(Backend,Opts)`

- *Description:* Ensure that the output directories for backend `Backend` are prepared.

get_autodoc_opts/3: PREDICATE

Usage: `get_autodoc_opts(Backend,Mod,Opts)`

- *Description:* Get the list of documentation options `Opts` for the `FileBase` file.
- *The following properties should hold at call time:*

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

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

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

autodoc_gen_doctree/5: PREDICATE

Usage: `autodoc_gen_doctree(Backend,FileBase,SourceSuffix,Opts,Mod)`

- *Description:* `FileBase` is the module specifier of the source file being documented (without extension, `SourceSuffix` is the suffix of the source). 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 `Backend` (the name of the output file also depends on `Backend`). The formats supported are given by `backend_id/1`.

- *Call and exit should be compatible with:*
  `Backend` is a supported backend. (`backend_id/1`)
  `FileBase` is the base name of a file (without extension). (`basename/1`)
  `SourceSuffix` is an atom. (`atm/1`)
  `Opts` is a list of `supported_options`. (`list/2`)
  `Mod` is an atom. (`atm/1`)

infodir_base/2: PREDICATE

No further documentation available for this predicate.

autodoc_compute_grefs/3: PREDICATE

Usage:

- *Description:* Compute the globally resolved references (including bibliography)

autodoc_translate_doctree/3: PREDICATE

Usage:

- *Description:* Translate the doctree using the specific backend

autodoc_finish/1: PREDICATE

No further documentation available for this predicate.

autodoc_gen_alternative/2: PREDICATE

No further documentation available for this predicate.

version_patch/2: PREDICATE

No further documentation available for this predicate.

version_date/2: PREDICATE

No further documentation available for this predicate.

version_numstr/2: PREDICATE

No further documentation available for this predicate.

15.3 Documentation on multifiles (`autodoc`)

autodoc_finish_hook/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_gen_alternative_hook/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

16 Documentation Abstract Syntax Tree

Author(s): Manuel Hermenegildo (original version), Jose F. Morales.

This module defines the intermediate tree representation `doctree/1` for documentation and its related operations.

Note: This part needs better documentation. – JFMC

16.1 Usage and interface (`autodoc_doctree`)

- **Library usage:**

  `:- use_module(library(autodoc_doctree)).`
- **Exports:**
  - *Predicates:*

    `cmd_type/1`, `doctree_is_empty/1`, `is_nonempty_doctree/1`, `empty_doctree/1`, `doclink_at/2`, `doclink_is_local/1`, `section_prop/2`, `section_select_prop/3`, `doctree_save/2`, `doctree_restore/2`, `doctree_simplify/2`, `doctree_putvars/5`, `doctree_scan_and_save_refs/2`, `doctree_prepare_refs_translate_and_write/3`, `doctree_to_rawtext/3`, `doctree_translate_and_write/3`, `escape_string/4`, `insert_printtoc/3`.
  - *Regular Types:*

    `doctree/1`, `doclink/1`, `doclabel/1`, `doctokens/1`.
  - *Multifiles:*

    `autodoc_rw_command_hook/4`, `autodoc_escape_string_hook/5`.
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(comments))`, `lpdocsrc(src(autodoc_refsdb))`, `lpdocsrc(src(autodoc_index))`, `lpdocsrc(src(autodoc_structure))`, `lpdocsrc(src(autodoc_filesystem))`, `lpdocsrc(src(autodoc_settings))`, `lpdocsrc(src(autodoc_texinfo))`, `lpdocsrc(src(autodoc_man))`, `lpdocsrc(src(autodoc_html))`.
  - *System library modules:*

    `write`, `operators`, `format`, `lists`, `make/system_extra`, `read`, `terms`, `make/make_rt`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

16.2 Documentation on exports (`autodoc_doctree`)

cmd_type/1: PREDICATE

No further documentation available for this predicate.

doctree/1: REGTYPE

Usage:

- *Description:* Intermediate tree representation for documentation

doctree_is_empty/1: PREDICATE

Usage: `doctree_is_empty(+R)`

- *Description:* Emptiness test
- *The following properties should hold at call time:*
  Intermediate tree representation for documentation (`doctree/1`)

is_nonempty_doctree/1: PREDICATE

Usage: `is_nonempty_doctree(+R)`

- *Description:* Not empty test
- *The following properties should hold at call time:*
  Intermediate tree representation for documentation (`doctree/1`)

empty_doctree/1: PREDICATE

Usage: `empty_doctree(-R)`

- *Description:* Empty
- *The following properties should hold at call time:*
  Intermediate tree representation for documentation (`doctree/1`)

doclink/1: REGTYPE

Usage:

- *Description:* A link to a document label

doclabel/1: REGTYPE

Usage:

- *Description:* An internal label

doclink_at/2: PREDICATE

No further documentation available for this predicate.

doclink_is_local/1: PREDICATE

No further documentation available for this predicate.

doctokens/1: REGTYPE

Usage:

- *Description:* Primitive doctree subset (ready for output, not further reducible)

section_prop/2: PREDICATE

No further documentation available for this predicate.

section_select_prop/3: PREDICATE

No further documentation available for this predicate.

doctree_save/2: PREDICATE

doctree_restore/2: PREDICATE

doctree_simplify/2: PREDICATE

No further documentation available for this predicate.

doctree_putvars/5: PREDICATE

Usage: `doctree_putvars(R0,DocSt,PDict,VarDict,R)`

- *Description:* Traverse `R0` and replace each `var(Name)` doctree item with a fresh variable `B`. For each replacement, the term `B=Var` is introduced in `VarDict`, where `Var` is the associated value to `Name` in the dictionary `PDict`.
- *The following properties should hold at call time:*

  Intermediate tree representation for documentation (`doctree/1`)

  `docstate(DocSt)` (`docstate/1`)
- *The following properties should hold upon exit:*

  Intermediate tree representation for documentation (`doctree/1`)

doctree_scan_and_save_refs/2: PREDICATE

Usage: `doctree_scan_and_save_refs(R,DocSt)`

- *Description:* Scan and save the references of the doctree
- *The following properties should hold at call time:*

  Intermediate tree representation for documentation (`doctree/1`)

  `docstate(DocSt)` (`docstate/1`)

doctree_prepare_refs_translate_and_write/3: PREDICATE

No further documentation available for this predicate.

doctree_to_rawtext/3: PREDICATE

Usage: `doctree_to_rawtext(X,DocSt,Y)`

- *Description:* `Y` is a simplified raw text representation of the `X`
- *Call and exit should be compatible with:*

  Intermediate tree representation for documentation (`doctree/1`)

  `docstate(DocSt)` (`docstate/1`)

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

doctree_translate_and_write/3: PREDICATE

No further documentation available for this predicate.

escape_string/4: PREDICATE

Usage:

- *Description:* Escapes needed characters in input string as needed for the target format.
- *The following properties should hold upon exit:*

  `Arg1` is currently instantiated to an atom. (`atom/1`)

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

  `docstate(Arg3)` (`docstate/1`)

  `Arg4` is a string (a list of character codes). (`string/1`)

insert_printtoc/3: PREDICATE

Usage: `insert_printtoc(R0,DocSt,R)`

- *Description:* Insert the command to print the table of contents in a given doctree. The right place may be different depending on the chosen backend.

16.3 Documentation on multifiles (`autodoc_doctree`)

autodoc_rw_command_hook/4: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_escape_string_hook/5: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

17 Handling the Document Structure

Author(s): Jose F. Morales.

17.1 Usage and interface (autodoc_structure)

- **Library usage:**

  `:- use_module(library(autodoc_structure)).`
- **Exports:**
  - *Predicates:*

    docstr_node/4, clean_docstr/0, parse_structure/0, check_deprecated_settings/0, parse_structure/2, parse_structure_list/2, is_list/1, standalone_docstr/1, get_mainmod/1, get_mainmod_spec/1, all_component_specs/1, component_spec/1.
- **Other modules used:**
  - *Application modules:*

    lpdocsrc(src(autodoc_settings)).
  - *System library modules:*

    filenames, aggregates, terms, make/make_rt.
  - *Internal (engine) modules:*

    term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

17.2 Documentation on exports (autodoc_structure)

docstr_node/4: PREDICATE

No further documentation available for this predicate.

The predicate is of type *data*.

clean_docstr/0: PREDICATE

No further documentation available for this predicate.

parse_structure/0: PREDICATE

No further documentation available for this predicate.

check_deprecated_settings/0: PREDICATE

No further documentation available for this predicate.

parse_structure_/2: PREDICATE

No further documentation available for this predicate.

parse_structure_list/2: PREDICATE

No further documentation available for this predicate.

is_list/1: PREDICATE

No further documentation available for this predicate.

standalone_docstr/1: PREDICATE

No further documentation available for this predicate.

get_mainmod/1: PREDICATE

No further documentation available for this predicate.

get_mainmod_spec/1: PREDICATE

No further documentation available for this predicate.

all_component_specs/1: PREDICATE

No further documentation available for this predicate.

component_spec/1: PREDICATE

No further documentation available for this predicate.

18 Access to Default Settings

Author(s): Jose F. Morales.

This module defines the setting values with some default values.

Note: This part needs better documentation. – JFMC

18.1 Usage and interface (`autodoc_settings`)

- **Library usage:**

  `:- use_module(library(autodoc_settings)).`
- **Exports:**
  - *Predicates:*

    `lpdoc_option/1`, `check_setting/1`, `setting_value_or_default/2`, `setting_value_or_default/3`, `setting_value/2`, `all_setting_values/2`, `get_command_option/1`, `requested_file_formats/1`, `load_vpaths/0`.
- **Other modules used:**
  - *System library modules:*

    `make/system_extra`, `make/make_rt`, `aggregates`, `distutils/distutils`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

18.2 Documentation on exports (`autodoc_settings`)

lpdoc_option/1: PREDICATE

Defines the global options of lpdoc.

The predicate is of type *data*.

check_setting/1: PREDICATE

No further documentation available for this predicate.

setting_value_or_default/2: PREDICATE

Usage: `setting_value_or_default(Var,Value)`

- *Description:* Returns in `Value` the value of the variable `Var`. In case this variable does not exists, it returns a default value. If there is no default value for the variable `Var` it fails.

setting_value_or_default/3: PREDICATE

No further documentation available for this predicate.

setting_value/2: PREDICATE

No further documentation available for this predicate.

all_setting_values/2: PREDICATE

No further documentation available for this predicate.

get_command_option/1: PREDICATE

No further documentation available for this predicate.

requested_file_formats/1: PREDICATE

Usage: `requested_file_formats(F)`

- *Description:* `F` is a requested file format

load_vpaths/0: PREDICATE

No further documentation available for this predicate.

LPdoc Backends

19 Texinfo Backend

Author(s): Manuel Hermenegildo, Jose F. Morales.

19.1 Usage and interface (`autodoc_texinfo`)

- **Library usage:**

  `:- use_module(library(autodoc_texinfo)).`
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_filesystem))`, `lpdocsrc(src(autodoc_structure))`, `lpdocsrc(src(autodoc_index))`, `lpdocsrc(src(autodoc_doctree))`, `lpdocsrc(src(autodoc_images))`, `lpdocsrc(src(autodoc_settings))`, `fastformat`, `lpdocsrc(src(comments))`, `lpdocsrc(makedir(LPDOCSETTINGS))`, `lpdocsrc(src(autodoc_aux))`.
  - *System library modules:*

    `lists`, `terms`, `format`, `messages`, `system`, `make/make_rt`, `file_utils`, `make/system_extra`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

19.2 Documentation on multifiles (`autodoc_texinfo`)

autodoc_escape_string_hook/5: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_rw_command_hook/4: PREDICATE

The predicate is *multifile*.

Usage: `autodoc_rw_command_hook(Backend,DocSt,Command,NewCommand)`

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

`Backend` is a supported backend.	(`backend_id/1`)
`docstate(DocSt)`	(`docstate/1`)
Intermediate tree representation for documentation	(`doctree/1`)
Intermediate tree representation for documentation	(`doctree/1`)

autodoc_finish_hook/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_gen_alternative_hook/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

20 HTML Backend

Author(s): Jose F. Morales.

20.1 Usage and interface (autodoc_html)

- **Library usage:**
  `:- use_module(library(autodoc_html)).`
- **Other modules used:**
  - *Application modules:*
    lpdocsrc(src(autodoc)), lpdocsrc(src(autodoc_structure)), lpdocsrc(src(autodoc_filesystem)), lpdocsrc(src(autodoc_doctree)), lpdocsrc(src(autodoc_index)), lpdocsrc(src(autodoc_refsdb)), lpdocsrc(src(autodoc_images)), lpdocsrc(src(autodoc_settings)), lpdocsrc(src(comments)), fastformat, lpdocsrc(src(autodoc_html_template)), lpdocsrc(src(distpkg_download)), lpdocsrc(src(autodoc_html_resources)).
  - *System library modules:*
    lists, dict, system, file_utils.
  - *Internal (engine) modules:*
    term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

20.2 Documentation on multifiles (autodoc_html)

autodoc_escape_string_hook/5: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_rw_command_hook/4: PREDICATE

The predicate is *multifile*.

Usage: `autodoc_rw_command_hook(Backend,DocSt,Command,NewCommand)`

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

`Backend` is a supported backend.	(`backend_id/1`)
`docstate(DocSt)`	(`docstate/1`)
Intermediate tree representation for documentation	(`doctree/1`)
Intermediate tree representation for documentation	(`doctree/1`)

autodoc_finish_hook/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_gen_alternative_hook/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

21 Resource Handling for the HTML Backend

Author(s): Jose F. Morales.

21.1 Usage and interface (`autodoc_html_resources`)

- **Library usage:**
  `:- use_module(library(autodoc_html_resources)).`
- **Exports:**
  - *Predicates:*
    `prepare_web_skel/1`, `prepare_mathjax/0`, `using_mathjax/1`.
- **Other modules used:**
  - *Application modules:*
    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_settings))`, `lpdocsrc(src(autodoc_filesystem))`.
  - *System library modules:*
    `messages`, `file_utils`, `make/system_extra`, `distutils/distutils`, `distutils/dirutils`, `terms`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

21.2 Documentation on exports (`autodoc_html_resources`)

prepare_web_skel/1: PREDICATE

No further documentation available for this predicate.

prepare_mathjax/0: PREDICATE

No further documentation available for this predicate.

using_mathjax/1: PREDICATE

No further documentation available for this predicate.

22 Template Support for the HTML Backend

Author(s): Jose F. Morales.

22.1 Usage and interface (`autodoc_html_template`)

- **Library usage:**
  `:- use_module(library(autodoc_html_template)).`
- **Exports:**
  - *Predicates:*
    `img_url/2`, `fmt_html_template/3`.
- **Other modules used:**
  - *Application modules:*
    `lpdocsrc(src(autodoc_settings))`.
  - *System library modules:*
    `messages`, `aggregates`, `system`, `file_utils`, `make/system_extra`, `distutils/distutils`, `distutils/dirutils`, `autoconfig/autoconfig`, `lists`, `terms`, `make/make_rt`, `pillow/html`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

22.2 Documentation on exports (`autodoc_html_template`)

img_url/2: PREDICATE

Usage: `img_url(Name,Url)`

- *Description:* Obtain the `URL` where image `Name` is or will be found.
- *Call and exit should be compatible with:*

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

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

fmt_html_template/3: PREDICATE

No further documentation available for this predicate.

23 Man Pages (man) Backend

Author(s): Jose F. Morales, Manuel Hermenegildo.

23.1 Usage and interface (`autodoc_man`)

- **Library usage:**

  `:- use_module(library(autodoc_man)).`
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_doctree))`, `lpdocsrc(src(autodoc_images))`, `lpdocsrc(src(autodoc_aux))`, `lpdocsrc(src(comments))`, `fastformat`.
  - *System library modules:*

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

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

23.2 Documentation on multifiles (`autodoc_man`)

autodoc_rw_command_hook/4: PREDICATE

The predicate is *multifile*.

Usage: `autodoc_rw_command_hook(Backend,DocSt,Command,NewCommand)`

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

  `Backend` is a supported backend. (`backend_id/1`)

  `docstate(DocSt)` (`docstate/1`)

  Intermediate tree representation for documentation (`doctree/1`)

  Intermediate tree representation for documentation (`doctree/1`)

autodoc_finish_hook/1: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

autodoc_gen_alternative_hook/2: PREDICATE

No further documentation available for this predicate.

The predicate is *multifile*.

24 Filesystem Abstraction

Author(s): Jose F. Morales.

This module provides definitions to assign unique file-system paths and names for each of the intermediate and final results of documentation generation.

24.1 Usage and interface (`autodoc_filesystem`)

- **Library usage:**

  `:- use_module(library(autodoc_filesystem)).`
- **Exports:**
  - *Predicates:*

    `file_format_name/2`, `supported_file_format/1`, `file_format_provided_by_backend/3`, `clean_fs_db/0`, `get_output_dir/2`, `get_cache_dir/2`, `ensure_output_dir/1`, `ensure_cache_dir/1`, `main_absfile_in_format/2`, `main_absfile_for_subtarget/3`, `absfile_for_aux/3`, `absfile_for_subtarget/4`, `main_output_name/1`, `get_subbase/3`, `absfile_to_relfile/3`, `clean_all/0`, `clean_docs_no_texi/0`, `clean_all_temporal/0`, `clean_intermediate/0`, `clean_tex_intermediate/0`.
  - *Regular Types:*

    `filename/1`, `basename/1`, `subtarget/1`.
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc_settings))`, `lpdocsrc(src(autodoc_structure))`, `lpdocsrc(src(autodoc))`.
  - *System library modules:*

    `aggregates`, `make/system_extra`, `terms`, `distutils/distutils`, `system`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

24.2 Documentation on exports (`autodoc_filesystem`)

filename/1: REGTYPE

Usage: `filename(X)`

- *Description:* `X` is the name of a file.

basename/1: REGTYPE

Usage: `basename(X)`

- *Description:* `X` is the base name of a file (without extension).

subtarget/1: REGTYPE

Usage:

– *Description:* The kind of intermediate/final results for a single documentation processing unit (module).

file_format_name/2: PREDICATE

No further documentation available for this predicate.

supported_file_format/1: PREDICATE

No further documentation available for this predicate.

file_format_provided_by_backend/3: PREDICATE

Usage: `file_format_provided_by_backend(Ext,Backend,Subtarget)`

– *Description:* `Backend` is the backend that generates files with format `Ext`

– *Call and exit should be compatible with:*

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

`Backend` is a supported backend. (`backend_id/1`)

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

clean_fs_db/0: PREDICATE

Usage:

– *Description:* Clean the internal state of the module.

get_output_dir/2: PREDICATE

Usage: `get_output_dir(Backend,Dir)`

– *Description:* Obtain the `Dir` directory where final documentation files will be stored

get_cache_dir/2: PREDICATE

Usage: `get_cache_dir(Backend,Dir)`

– *Description:* Obtain the `Dir` directory where final documentation files will be stored

ensure_output_dir/1: PREDICATE

No further documentation available for this predicate.

ensure_cache_dir/1: PREDICATE

No further documentation available for this predicate.

main_absfile_in_format/2: PREDICATE

Usage: `main_absfile_in_format(Ext,File)`

- *Description:* `File` is the absolute file name for the documentation in `Ext` format of the *main* module

main_absfile_for_subtarget/3: PREDICATE

No further documentation available for this predicate.

absfile_for_aux/3: PREDICATE

Usage: `absfile_for_aux(AuxName,Backend,AbsFile)`

- *Description:* Absolute file for an auxiliary output file (e.g. CSS, images, etc.)

absfile_for_subtarget/4: PREDICATE

No further documentation available for this predicate.

main_output_name/1: PREDICATE

No further documentation available for this predicate.

get_subbase/3: PREDICATE

Usage: `get_subbase(Base,Sub,SubBase)`

- *Description:* `SubBase` is the name for the sub-file (`Sub`) associated with `Base`
- *The following properties should hold upon exit:*

  `Base` is the base name of a file (without extension). (`basename/1`)

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

  `SubBase` is the base name of a file (without extension). (`basename/1`)

absfile_to_relfile/3: PREDICATE

Usage: `absfile_to_relfile(A,Backend,B)`

- *Description:* Obtain the relative path, w.r.t. the output directory, of an absolute file. This is useful, e.g., for URLs.

clean_all/0: PREDICATE

No further documentation available for this predicate.

clean_docs_no_texi/0: PREDICATE

No further documentation available for this predicate.

clean_all_temporal/0: PREDICATE

No further documentation available for this predicate.

clean_intermediate/0: PREDICATE

No further documentation available for this predicate.

clean_tex_intermediate/0: PREDICATE

No further documentation available for this predicate.

25 Indexing Commands (Definition and Formatting)

Author(s): Jose F. Morales.

This module defines index commands and formatting.

Note: This part needs better documentation. – JFMC

25.1 Usage and interface (`autodoc_index`)

- **Library usage:**

  `:- use_module(library(autodoc_index)).`
- **Exports:**
  - *Predicates:*

    `get_idxsub/2`, `get_idxbase/3`, `typeindex/5`, `idx_get_indices/3`, `is_index_cmd/1`, `codetype/1`, `normalize_index_cmd/3`, `fmt_idx_env/7`, `fmt_index/4`.
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_filesystem))`, `lpdocsrc(src(autodoc_doctree))`, `lpdocsrc(src(autodoc_structure))`, `lpdocsrc(src(autodoc_refsdb))`.
  - *System library modules:*

    `dict`, `lists`, `aggregates`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

25.2 Documentation on exports (`autodoc_index`)

get_idxsub/2: PREDICATE

No further documentation available for this predicate.

get_idxbase/3: PREDICATE

No further documentation available for this predicate.

typeindex/5: PREDICATE

Usage: `typeindex(Type,Index,IType,Name,Comment)`

- *Description:* `Index` is the (info) index name in which objects of type `Type` go. `Name` is the title of the index in the documentation. `IType` is the type of index; an empty string means normal. code`Comment` is a comment to include before the index.

- *The following properties should hold upon exit:*

`Type` is currently instantiated to an atom.	(`atom/1`)
`Index` is an atom.	(`atm/1`)
`IType` is a string (a list of character codes).	(`string/1`)
`Name` is a string (a list of character codes).	(`string/1`)
Intermediate tree representation for documentation	(`doctree/1`)

idx_get_indices/3: PREDICATE

No further documentation available for this predicate.

is_index_cmd/1: PREDICATE

No further documentation available for this predicate.

codetype/1: PREDICATE

No further documentation available for this predicate.

normalize_index_cmd/3: PREDICATE

No further documentation available for this predicate.

fmt_idx_env/7: PREDICATE

No further documentation available for this predicate.

fmt_index/4: PREDICATE

No further documentation available for this predicate.

26 Database of Documentation References

Author(s): Jose F. Morales.

This module stores and manages all the documentation references (indices, sections, bibliography, etc.)

26.1 Usage and interface (`autodoc_refsdb`)

- **Library usage:**

  `:- use_module(library(autodoc_refsdb)).`
- **Exports:**
  - *Predicates:*

    `labgen_init/1`, `labgen_clean/1`, `labgen_get/2`, `refs_clean/1`, `add_refs_entry/2`, `save_refs/1`, `refs_closure_entry/3`, `refs_closure_clean/1`, `compute_refs_and_biblio/1`, `prepare_current_refs/1`, `clean_current_refs/1`, `secttree_resolve/3`, `get_main_title/2`.
  - *Regular Types:*

    `secttree/1`.
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_doctree))`, `lpdocsrc(src(autodoc_structure))`, `lpdocsrc(src(autodoc_filesystem))`, `lpdocsrc(src(autodoc_bibrefs))`, `.(autodoc_structure)`.
  - *System library modules:*

    `write`, `read`, `system`, `aggregates`, `lists`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

26.2 Documentation on exports (`autodoc_refsdb`)

labgen_init/1: PREDICATE

No further documentation available for this predicate.

labgen_clean/1: PREDICATE

No further documentation available for this predicate.

labgen_get/2: PREDICATE

No further documentation available for this predicate.

refs_clean/1: PREDICATE

No further documentation available for this predicate.

add_refs_entry/2: PREDICATE

No further documentation available for this predicate.

save_refs/1: PREDICATE

No further documentation available for this predicate.

refs_closure_entry/3: PREDICATE

No further documentation available for this predicate.

The predicate is of type *data*.

refs_closure_clean/1: PREDICATE

No further documentation available for this predicate.

compute_refs_and_biblio/1: PREDICATE

No further documentation available for this predicate.

prepare_current_refs/1: PREDICATE

Usage:

- *Description:* Prepare references for the translation of the current file `foo`
- *Call and exit should be compatible with:*

  `docstate(Arg1)` (`docstate/1`)

clean_current_refs/1: PREDICATE

Usage:

- *Description:* Clean the data stored by `prepare_current_refs/1`.
- *Call and exit should be compatible with:*

  `docstate(Arg1)` (`docstate/1`)

secttree/1: REGTYPE

Usage:

- *Description:* A tree of sections

secttree_resolve/3: PREDICATE

Usage: `secttree_resolve(LabelName,Tree,Link)`

- *Description:* Locate in the section tree `Tree` the section with label name `LabelName` and return the `Link` to the section.
- *Call and exit should be compatible with:*

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

  Intermediate tree representation for documentation (`doctree/1`)

  A link to a document label (`doclink/1`)

get_main_title/2: PREDICATE

Usage: `get_main_title(MainTitleR,DocSt)`

- *Description:* `MainTitleR` is the title of the main section for the whole document
- *The following properties should hold upon exit:*

  Intermediate tree representation for documentation (`doctree/1`)

  `docstate(DocSt)` (`docstate/1`)

27 Error Messages

Author(s): Manuel Hermenegildo.

27.1 Usage and interface (`autodoc_errors`)

- **Library usage:**
  `:- use_module(library(autodoc_errors)).`
- **Exports:**
  - *Predicates:*
    `error_text/3`.
- **Other modules used:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

27.2 Documentation on exports (`autodoc_errors`)

error_text/3: PREDICATE

No further documentation available for this predicate.

28 Resolution of Bibliographical References

Author(s): Manuel Hermenegildo (original version), Jose F. Morales.

This module provides a predicate to resolve the bibliographical references found during the generation of documentation.

28.1 Usage and interface (`autodoc_bibrefs`)

- **Library usage:**
  `:- use_module(library(autodoc_bibrefs)).`
- **Exports:**
  - *Predicates:*
    `resolve_bibliography/1`.
- **Other modules used:**
  - *Application modules:*
    `lpdocsrc(makedir(LPDOCSETTINGS))`, `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_doctree))`, `lpdocsrc(src(autodoc_refsdb))`, `lpdocsrc(src(autodoc_aux))`, `lpdocsrc(src(autodoc_settings))`, `lpdocsrc(src(autodoc_parse))`.
  - *System library modules:*
    `dict`, `aggregates`, `terms`, `file_utils`, `lists`, `format`, `make/make_rt`, `make/system_extra`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

28.2 Documentation on exports (`autodoc_bibrefs`)

resolve_bibliography/1: PREDICATE

Usage: `resolve_bibliography(DocSt)`

- *Description:* This predicate resolves bibliographical references. The algorithm is as follows:
  - Write all the bibliographical references to a `.aux` file.
  - Invoke BibTeX with a customized `.bst` file that generates a pseudo-docstring.
  - Load the docstring and fix its syntax.
  - Parse the docstring as a doctree.
  - Extract `(Label,Ref)` pairs from `bibitem` commands.

  Both the docstring and label/reference pairs are kept in the `DocSt`, and used later to map symbolic references to textual labels.
- *The following properties should hold at call time:*
  `docstate(DocSt)` (`docstate/1`)

29 Auxiliary Definitions

Author(s): Manuel Hermenegildo, Jose F. Morales.

29.1 Usage and interface (autodoc_aux)

- **Library usage:**

  `:- use_module(library(autodoc_aux)).`
- **Exports:**
  - *Predicates:*

    `read_file/2`, `ascii_blank_lines/2`, `sh_exec/2`.
- **Other modules used:**
  - *Application modules:*

    `lpdocsrc(src(autodoc_settings))`.
  - *System library modules:*

    `messages`, `system`, `make/system_extra`, `lists`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

29.2 Documentation on exports (autodoc_aux)

read_file/2: PREDICATE

No further documentation available for this predicate.

ascii_blank_lines/2: PREDICATE

No further documentation available for this predicate.

sh_exec/2: PREDICATE

No further documentation available for this predicate.

30 Image Handling

Author(s): Jose F. Morales.

This module defines the handling of image commands. It defines predicates to locate and convert images in the different formats required for documentation.

Note: This part needs better documentation. – JFMC

30.1 Usage and interface (`autodoc_images`)

- **Library usage:**
  `:- use_module(library(autodoc_images)).`
- **Exports:**
  - *Predicates:*
    `locate_and_convert_image/4`.
- **Other modules used:**
  - *Application modules:*
    `lpdocsrc(src(autodoc))`, `lpdocsrc(src(autodoc_filesystem))`, `lpdocsrc(src(autodoc_settings))`, `lpdocsrc(src(autodoc_aux))`, `lpdocsrc(makedir(LPDOCSETTINGS))`.
  - *System library modules:*
    `terms`, `make/make_rt`, `make/system_extra`, `system`, `errhandle`, `messages`, `format`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `mattr_global`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.

30.2 Documentation on exports (`autodoc_images`)

locate_and_convert_image/4: PREDICATE

Usage:
`locate_and_convert_image(SrcSpecS,AcceptedFormats,DocSt,TargetFileS)`

- *Description:* The image at `SrcSpecS` is located (as one of the known formats `known_format/1`) and converted to one of the `AcceptedFormats`. The target file is called `TargetFileS`
- *Call and exit should be compatible with:*
  `SrcSpecS` is a string (a list of character codes). (`string/1`)
  `AcceptedFormats` is a list of `atoms`. (`list/2`)
  `docstate(DocSt)` (`docstate/1`)
  `TargetFileS` is a string (a list of character codes). (`string/1`)

References

[Bue95] F. Bueno.
The CIAO Multiparadigm Compiler: A User's Manual.
Technical Report CLIP8/95.0, Facultad de Informática, 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. López-García, 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. López-García, 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 Informática, 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. López-García, 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, 21(4–6):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 Informática, 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.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

Library/Module Definition Index

A

assertions 35
assertions_props 45
autodoc 105
autodoc_aux 147
autodoc_bibrefs 145
autodoc_doctree 113
autodoc_errors 143
autodoc_filesystem 133
autodoc_html 125
autodoc_html_resources 127
autodoc_html_template 129
autodoc_images 149
autodoc_index 137
autodoc_man 131
autodoc_refsdb 139
autodoc_settings 119
autodoc_structure 117
autodoc_texinfo 123

B

basic_props 55

C

comments 19

E

example_module 89

M

meta_props 81

N

native_props 71

R

regtypes 51
rtchecks 97

U

unittest 99

Predicate/Method Definition Index

A

absfile_for_aux/3 135
absfile_for_subtarget/4 135
absfile_to_relfile/3 135
add_refs_entry/2 140
all_component_specs/1 118
all_indices/2 110
all_setting_values/2 120
ascii_blank_lines/2 147
autodoc_compute_grefs/3 111
autodoc_escape_string_hook/5 .... 116, 123, 125
autodoc_finish/1 111
autodoc_finish_hook/1 .... 112, 123, 125, 131
autodoc_gen_alternative/2 111
autodoc_gen_alternative_hook/2 .... 112, 124, 126, 131
autodoc_gen_doctree/5 110
autodoc_rw_command_hook/4 .... 116, 123, 125, 131
autodoc_translate_doctree/3 111

B

backend_alt_format/2 107
backend_ignores_components/1 107

C

callme/2 82
check/1 42
check_deprecated_settings/0 117
check_setting/1 119
clean_all/0 135
clean_all_temporal/0 135
clean_current_refs/1 140
clean_docs_no_texi/0 135
clean_docstr/0 117
clean_fs_db/0 134
clean_intermediate/0 136
clean_tex_intermediate/0 136
cmd_type/1 113
codetype/1 138
component_spec/1 118
compute_refs_and_biblio/1 140

D

doc_customdic_get/3 109
doc_customdic_lookup/3 109
doc_customdic_maybe_get/3 109
doc_customdic_replace/4 109
doc_doing_mainmod/1 109
doc_id_type/3 26
doc_message/2 109
doc_message/3 109
doc_modname/2 109
doc_opt/2 109
doclink_at/2 114
doclink_is_local/1 114
docstate_backend/2 108
docstate_currmod/2 108
docstate_has_index/2 109
docstate_inputfile/2 109
docstate_opts/2 108
docstate_set_currmod/3 108
docstate_set_customdic/3 109
docstate_set_opts/3 108
docstr_node/4 117
doctree_is_empty/1 114
doctree_prepare_refs_translate_and_write/3 115
doctree_putvars/5 115
doctree_restore/2 115
doctree_save/2 115
doctree_scan_and_save_refs/2 115
doctree_simplify/2 115
doctree_to_rawtext/3 115
doctree_translate_and_write/3 116

E

empty_doctree/1 114
ensure_cache_dir/1 134
ensure_output_dir/1 134
ensure_output_dir_prepared/2 110
error_text/3 143
escape_string/4 116

F

false/1 43
file_format_name/2 134
file_format_provided_by_backend/3 134
fmt_html_template/3 129
fmt_idx_env/7 138
fmt_index/4 138

G

get_autodoc_opts/3 . . . 110
get_cache_dir/2 . . . 134
get_command_option/1 . . . 120
get_doc/4 . . . 110
get_idxbase/3 . . . 137
get_idxsub/2 . . . 137
get_main_title/2 . . . 141
get_mainmod/1 . . . 118
get_mainmod_spec/1 . . . 118
get_output_dir/2 . . . 134
get_subbase/3 . . . 135

I

idx_get_indices/3 . . . 138
img_url/2 . . . 129
index_comment/2 . . . 107
infodir_base/2 . . . 111
insert_printtoc/3 . . . 116
is/2 . . . 96
is_index_cmd/1 . . . 138
is_list/1 . . . 118
is_nonempty_doctree/1 . . . 114

L

labgen_clean/1 . . . 139
labgen_get/2 . . . 139
labgen_init/1 . . . 139
load_vpaths/0 . . . 120
locate_and_convert_image/4 . . . 149
lpdoc_option/1 . . . 119

M

main_absfile_for_subtarget/3 . . . 135
main_absfile_in_format/2 . . . 134
main_output_name/1 . . . 135
mytype/1 . . . 92

N

normalize_index_cmd/3 . . . 138

O

option_comment/2 . . . 107

P

p/1 . . . 91
p/3 . . . 94
p/5 . . . 91
parse_structure/0 . . . 117
parse_structure_/2 . . . 117
parse_structure_list/2 . . . 118
prepare_current_refs/1 . . . 140
prepare_mathjax/0 . . . 127
prepare_web_skel/1 . . . 127

Q

q/1 . . . 93
q/2 . . . 90

R

r/1 . . . 90
read_file/2 . . . 147
refs_clean/1 . . . 139
refs_closure_clean/1 . . . 140
refs_closure_entry/3 . . . 140
requested_file_formats/1 . . . 120
reset_output_dir_db/0 . . . 110
resolve_bibliography/1 . . . 145

S

s/1 . . . 92
save_refs/1 . . . 140
section_prop/2 . . . 115
section_select_prop/3 . . . 115
secttree_resolve/3 . . . 140
setting_value/2 . . . 120
setting_value_or_default/2 . . . 119
setting_value_or_default/3 . . . 119
sh_exec/2 . . . 147
standalone_docstr/1 . . . 118
supported_file_format/1 . . . 134

T

t/5 . . . 92
top_suffix/2 . . . 107
true/1 . . . 43
trust/1 . . . 42
typeindex/5 . . . 137

U

u/3 91
using_mathjax/1 127

V

version_date/2 111
version_numstr/2 111
version_patch/2 111

W

w/1 92

Property Definition Index

B

bind_ins/1 69

C

call/2 81
clique/1 71
clique_1/1 72
compat/2 66
constraint/1 72
covered/1 72
covered/2 72

D

deprecated/1 67
docstring/1 .... 19, 50

E

equiv/2 69
error_free/1 69
eval/1 69
exception/1 72
exception/2 72

F

fails/1 73
filter/2 69
finite_solutions/1 73

H

have_choicepoints/1 73
head_pattern/1 46

I

indep/1 73
indep/2 73
inst/2 66
instance/2 80
intervals/2 80
is_det/1 73
iso/1 67

L

linear/1 74
long/1 91

M

member/2 63
memo/1 69
mshare/1 74
mut_exclusive/1 74

N

nabody/1 48
native/1 68
native/2 68
no_choicepoints/1 74
no_exception/1 75
no_exception/2 75
no_rtcheck/1 68
no_signal/1 75
no_signal/2 75
non_det/1 75
nonground/1 75
not_covered/1 75
not_fails/1 75
not_further_inst/2 67
not_mut_exclusive/1 76
num_solutions/2 76

P

pe_type/1 70
possibly_fails/1 76
possibly_nondet/1 77
prop/2 81
prop_abs/1 82

R

regtype/1 68
regtype/2 82
relations/2 77

S

sideff/2 68
sideff_hard/1 77
sideff_pure/1 77
sideff_soft/1 77
signal/1 77
signal/2 77
signals/2 78
size/2 78
size/3 78
size_lb/2 78
size_metric/3 78
size_metric/4 78
size_o/2 78
size_ub/2 78
solutions/2 76
steps/2 79
steps_lb/2 79
steps_o/2 79
steps_ub/2 79
stringcommand/1 20
supported_option/1 107

T

tau/1 79
terminates/1 80
test_type/2 80
throws/2 80

U

user_output/2 80

Regular Type Definition Index

A

aorb/1 89
assrt_body/1 45
assrt_status/1 50
assrt_type/1 50
atm/1 58
atm_or_atm_list/1 66

B

backend_id/1 106
bar/1 89
basename/1 133
baz/1 89

C

c_assrt_body/1 48
callable/1 60
character_code/1 64
complex_arg_property/1 46
complex_goal_property/1 47
constant/1 60

D

dictionary/1 48
doclabel/1 114
doclink/1 114
docstate/1 108
doctokens/1 114
doctree/1 113

F

filename/1 133
filetype/1 26
flag_values/1 69
flt/1 57

G

g_assrt_body/1 49
gnd/1 59
gndstr/1 59

I

int/1 56

Y

ymd_date/1 32

L

list/1 .... 61, 93
list/2 .... 62, 95
list_or_aorb/2 90

M

modtype/1 110

N

nlist/2 62
nnegint/1 56
num/1 57
num_code/1 65

O

operator_specifier/1 61

P

predfunctor/1 50
predname/1 65
property_conjunction/1 47
property_starterm/1 47
propfunctor/1 50

S

s_assrt_body/1 48
secttree/1 140
sequence/2 63
sequence_or_list/2 64
string/1 65
struct/1 58
subtarget/1 133

T

term/1 55
time_struct/1 32
tree_of/2 90

V

version_descriptor/1 25
version_maintenance_type/1 32
version_number/1 32

Declaration Definition Index

C

calls/1 37
calls/2 37
comment/2 42
comp/1 39
comp/2 39

D

decl/1 41
decl/2 41
doc/2 .... 26, 41

E

entry/1 40
exit/1 40
exit/2 41

M

modedef/1 41

P

pred/1 36
pred/2 37
prop/1 39
prop/2 40

R

regtype/1 54
regtype/2 54

S

success/1 38
success/2 38

T

test/1 38
test/2 38
texec/1 37
texec/2 37

Concept Definition Index

.

.bib files 10, 21

@

@! command 24
@’ command 24
@. command 24
@.. command 24
@= command 24
@? command 24
@‘ command 24
@~ command 24
@" command 24
@^ command 24
@aa command 24
@AA command 24
@ae command 24
@AE command 24
@apl command 21
@author command 22
@b command 24
@begin{alert} command 23
@begin{cartouche} command 23
@begin{description} command 22
@begin{displaymath} command 23
@begin{enumerate} command 22
@begin{itemize} command 22
@begin{verbatim} command 23
@bf command 23
@bullet command 25
@c command 24
@cindex command 20
@cite command 21
@comment command 22
@concept command 20
@copyright command 25
@d command 24
@decl command 21
@defmathcmd/2 command 24
@defmathcmd/3 command 24
@em command 23
@email command 22
@end{alert} command 23
@end{cartouche} command 23
@end{description} command 22
@end{displaymath} command 24
@end{enumerate} command 22
@end{itemize} command 22
@end{verbatim} command 23
@file command 21
@footnote command 23
@H command 24
@hfill command 23
@i command 25
@image command 25
@include command 25
@includedef command 25
@includefact command 25
@includeverbatim command 25
@index command 20
@iso command 25
@item command 22
@j command 25
@key command 23
@l command 24
@L command 24
@lib command 21
@math command 23
@noindent command 23
@o command 24
@O command 24
@oe command 24
@OE command 24
@op command 21
@p command 23
@pred command 21
@ref command 22
@result command 25
@section command 23
@sp command 23
@ss command 24
@subsection command 23
@t command 24
@today command 23
@tt command 23
@u command 24
@uref command 22
@v command 24
@var command 21

A

A4 paper 11
abstract 27
accents 24
acceptable modes 46
acknowledgements 27
appendix 28
application 4

assertion body syntax . . . 45, 48, 49
assertions . . . 19
author . . . 27
automatic documentation . . . 105
automatic documentation library . . . 105
avoiding indentation . . . 23

B

bibliographic citations . . . 21
bibliographic entries . . . 10
bibliographic entry . . . 21
bibtex . . . 10, 21
blank lines . . . 23
bold face . . . 23
bug . . . 29

C

calls assertion . . . 37, 38
check assertion . . . 42
Ciao . . . 9
comment . . . 22
comment assertion . . . 42
comments, machine readable . . . 35
comp assertion . . . 39
compatibility properties . . . 51
component files . . . 4
contents area . . . 12
copyright . . . 27

D

date . . . 23
decl assertion . . . 41
description list . . . 22
document structure . . . 10
documentation format . . . 105
documentation strings . . . 26

E

emacs Ciao mode . . . 30
Emacs, accessing info files . . . 13
Emacs, generating manuals from . . . 9
Emacs, LPdoc mode . . . 9
email address . . . 22
email addresses . . . 21
emphasis face . . . 23
encapsulated postscript . . . 25
entry assertion . . . 40
enumerated list . . . 22
escape sequences . . . 20
example of lpdoc use . . . 83
exit assertion . . . 40

F

false assertion . . . 43
fixed format text . . . 23
fixed-width font . . . 23
footnote . . . 23
formatting commands . . . 19, 35
framed box . . . 23

G

generating from Emacs . . . 9
generating manuals . . . 9

H

hard side-effects . . . 77
html index page . . . 12

I

image file . . . 25
images, inserting . . . 25
images, scaling . . . 25
including a predicate definition . . . 25
including an image . . . 25
including code . . . 25
including files . . . 25
including images . . . 25
including or not authors . . . 10
including or not bug info . . . 10
including or not changelog . . . 10
including or not versions, patches . . . 10
indentation, avoiding . . . 23
index pages out of order . . . 12
info path list . . . 14
installation . . . 3
installation, of manuals . . . 12
instantiation properties . . . 51
internals manual . . . 16
introduction . . . 28
italics face . . . 23
item in an itemized list . . . 22
itemized list . . . 22

K

keyboard key 23

L

LaTeX notation 23
letter size paper 11
library 3
literate programming 13
log of changes 29

M

machine readable comments 26
main body 28
main file 4
Makefile 101
module comment 28
module declaration 105

N

new item in description list 22

O

one-sided printing 10

P

page numbering, changing 10
page size, changing 11
page style, changing 11
paragraph break 23
parametric property 82
parametric regular type abstractions 82
parametric type functor 54
parts in a large document 17
parts in large documents 31
planned improvement 29
pred assertion 36, 37
program section 28, 29
program subsection 29
Prolog, Ciao 9
prop assertion 39, 40
properties of computations 51
properties of execution states 51
properties, basic 55
properties, native 71
property abstraction 82

R

references 21
regtype assertion 54
regular type expression 54
running unit tests 99

S

section 23
sections 21
SETTINGS.pl 9
sharing pieces of text 25
sharing sets 74
soft side-effects 77
space, extra lines 23
spcae, horizontal fill 23
special characters 24
strong face 23
subsection 23
subtitle 27
success assertion 38
supported documentation formats 110
synopsis section of the man page 10
syntax of formatting commands 20
system modules 10

T

test assertion 38, 39, 99
texec assertion 37
texinfo 105
texinfo files 4
textual comments 19
thesis-like style 11
title 26
true assertion 43
trust assertion 42
two-sided 10
typewriter-like font 23

U

unit tests 99
Universal Resource Locator 22
URL 22
urls 21
usage of a command 25
usage of the application 10

V

verbatim text 23
version maintenance mode for packages 30
version number 29

W

WWW address 22

Author Index

A

Anonymous Author 1 89
Anonymous Author 2 89

D

Daniel Cabeza 55

E

Edison Mera 71, 97, 99

F

Francisco Bueno 35, 51, 71, 81

G

German Puebla 35

J

Jose F. Morales 105, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 145, 147, 149

M

Manuel Hermenegildo . . . 9, 19, 35, 45, 51, 55, 71, 83, 101, 105, 113, 123, 131, 143, 145, 147

P

Pedro Lopez 51, 71

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.

,

’,’/2 63

(

(prop)/2 82

*

*/2 47

.

.(autodoc_structure) 139
.bib files 10, 21
.cshrc 12
.profile 12

:

::/2 36

=

= /2 26, 27, 28, 29, 30, 31, 32
=>/2 36

@

@! command 24
@’ command 24
@. command 24
@.. command 24
@= command 24
@? command 24
@‘ command 24
@~ command 24
@" command 24
@^ command 24
@aa command 24
@AA command 24
@ae command 24
@AE command 24
@apl command 21
@author command 22
@b command 24
@begin{alert} command 23
@begin{cartouche} command 23
@begin{description} command 22
@begin{displaymath} command 23
@begin{enumerate} command 22
@begin{itemize} command 22
@begin{verbatim} command 23
@bf command 23
@bullet command 25
@c command 24
@cindex command 20
@cite command 10, 21
@comment command 22
@concept command 20
@copyright command 25
@d command 24
@decl command 21
@defmathcmd/2 command 24
@defmathcmd/3 command 24
@em command 23
@email command 22
@end{alert} command 23
@end{cartouche} command 23
@end{description} command 22
@end{displaymath} command 24
@end{enumerate} command 22
@end{itemize} command 22
@end{verbatim} command 23
@file command 21
@footnote command 23
@H command 24
@hfill command 23
@i command 25
@image command 25
@include command 25
@includedef command 25
@includefact command 25
@includeverbatim command 25
@index command 20
@iso command 25
@item command 22
@j command 25
@key command 23
@l command 24
@L command 24
@lib command 21
@math command 23
@noindent command 23
@o command 24

@O command . . . 24
@oe command . . . 24
@OE command . . . 24
@op command . . . 21
@p command . . . 23
@pred command . . . 21
@ref command . . . 22
@result command . . . 25
@section command . . . 23
@sp command . . . 23
@ss command . . . 24
@subsection command . . . 23
@t command . . . 24
@today command . . . 23
@tt command . . . 23
@u command . . . 24
@uref command . . . 22
@v command . . . 24
@var command . . . 21

+

+/1 . . . 46
+/2 . . . 46

A

A4 paper . . . 11
absfile_for_aux/3 . . . 133, 135
absfile_for_subtarget/4 . . . 133, 135
absfile_to_relfile/3 . . . 133, 135
abstract . . . 27
accents . . . 24
acceptable modes . . . 46
acknowledgements . . . 27
add_refs_entry/2 . . . 139, 140
address . . . 22
aggregates . . . 106, 117, 119, 129, 133, 137, 139, 145
all_component_specs/1 . . . 117, 118
all_indices/2 . . . 106, 110
all_setting_values/2 . . . 119, 120
analyzer output . . . 43
Anonymous Author 1 . . . 89
Anonymous Author 2 . . . 89
aorb/1 . . . 89
appendix . . . 28
application . . . 4
arithexpression/1 . . . 96
arithmetic . . . 19, 36, 37, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
ascii_blank_lines/2 . . . 147
assertion body syntax . . . 45, 48, 49
assertion status . . . 37, 38, 39, 41
assertions . . . 3, 16, 17, 19, 35, 36, 45, 105
assertions/assertions_props . . . 36, 53, 106
assertions/assrt_lib . . . 106
assertions/native_props . . . 55, 89
assertions_props . . . 45
assrt_body/1 . . . 36, 37, 40, 41, 45, 54
assrt_status/1 . . . 37, 38, 39, 40, 41, 45, 50, 54
assrt_type/1 . . . 45, 50
atm/1 . . . 32, 55, 58, 110, 111, 129, 134, 135, 138
atm_or_atm_list/1 . . . 55, 66
atom/1 . . . 107, 116, 138
atomic_basic . . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
attributes . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
author . . . 27
author indexing . . . 27
autoconfig/autoconfig . . . 129
autodoc . . . 3, 10, 105
autodoc_aux . . . 147
autodoc_bibrefs . . . 145
autodoc_compute_grefs/3 . . . 106, 111
autodoc_doctree . . . 113
autodoc_errors . . . 143
autodoc_escape_string_hook/5 . . 113, 116, 123, 125
autodoc_filesystem . . . 133
autodoc_finish/1 . . . 106, 111
autodoc_finish_hook/1 . . . 106, 112, 123, 125, 131
autodoc_gen_alternative/2 . . . 106, 111
autodoc_gen_alternative_hook/2 . . . 106, 112, 124, 126, 131
autodoc_gen_doctree/5 . . . 106, 110
autodoc_html . . . 105, 125
autodoc_html_resources . . . 127
autodoc_html_template . . . 129
autodoc_images . . . 149
autodoc_index . . . 137
autodoc_man . . . 105, 131
autodoc_refsdb . . . 139

autodoc_rw_command_hook/4 . . . . . 113, 116, 123, 125, 131
autodoc_settings . . . 119
autodoc_structure . . . 117
autodoc_texinfo . . . 105, 123
autodoc_translate_doctree/3 . . . 106, 111
autodocformats . . . 3
automatic documentation . . . 105
automatic documentation library . . . 3, 105
avoiding indentation . . . 23

B

backend_alt_format/2 . . . 106, 107
backend_id/1 . . . 106, 110, 111, 123, 125, 131, 134
backend_ignores_components/1 . . . 106, 107
bar/1 . . . 89, 94
basename/1 . . . 111, 133, 135
bash . . . 12
basic_props . . . 3, 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
basic_props.pl . . . 97
basic_props:regtype/1 . . . 51
basiccontrol . . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
baz(Arg3) . . . 94
baz/1 . . . 89, 94
bibliographic citations . . . 21
bibliographic entries . . . 10
bibliographic entry . . . 21
bibtex . . . 10, 21, 102
bind_ins/1 . . . 55, 57, 63, 69
blank lines . . . 23
bold face . . . 23
bug . . . 29

C

c_assrt_body/1 . . . 37, 38, 40, 45, 48
call(T,A) . . . 96
call/1 . . . 48
call/2 . . . 81
callable/1 . . . 55, 60, 76, 81, 82
callme/2 . . . 81, 82
calls assertion . . . 37, 38
calls/1 . . . 36, 37, 38, 40
calls/2 . . . 36, 37
character string . . . 35
character_code/1 . . . 55, 64, 65
check assertion . . . 42
Check(X) . . . 76
check/1 . . . 36, 42, 43
check_deprecated_settings/0 . . . 117
check_setting/1 . . . 119
Ciao . . . 3, 9, 13, 14, 83, 101
Ciao Emacs mode . . . 9
ciaopp . . . 71
cjpeg . . . 102
clean_all/0 . . . 133, 135
clean_all_temporal/0 . . . 133, 135
clean_current_refs/1 . . . 139, 140
clean_docs_no_texi/0 . . . 133, 135
clean_docstr/0 . . . 117
clean_fs_db/0 . . . 133, 134
clean_intermediate/0 . . . 133, 136
clean_tex_intermediate/0 . . . 133, 136
clique/1 . . . 71
clique_1/1 . . . 71, 72
CLP . . . 3
cmd_type/1 . . . 113
codetype/1 . . . 137, 138
comment . . . 22
comment assertion . . . 42
comment string . . . 46, 48, 49
comment/2 . . . 36, 42
comments . . . 3, 17, 19
comments, machine readable . . . 35
CommentType=ack . . . 27
CommentType=appendix . . . 28
CommentType=author . . . 27
CommentType=bug . . . 29
CommentType=copyright . . . 27
CommentType=doinclude . . . 30, 31
CommentType=filetype . . . 31
CommentType=hide . . . 31
CommentType=module . . . 28
CommentType=nodoc . . . 32
CommentType=section . . . 28
CommentType=subsection . . . 29
CommentType=subtitle . . . 27
CommentType=summary . . . 28
CommentType=title . . . 26
CommentType=usage . . . 28
CommentType=version_maintenance . . . 30

comp assertion . . . 39
comp/1 . . . 36, 39, 49
comp/2 . . . 36, 39
compat/2 . . . 55, 66
compatibility properties . . . 51
compatible . . . 45
compiler/c_itf . . . 106
compiler/compiler . . . 106
complex argument property . . . 45, 46, 47, 48, 49
complex goal property . . . 46, 48, 49
complex_arg_property/1 . . . 45, 46, 48, 49
complex_goal_property/1 . . . 45, 46, 47, 49
component files . . . 4
component_spec/1 . . . 117, 118
compute_refs_and_biblio/1 . . . 139, 140
constant/1 . . . 55, 60
constraint/1 . . . 71, 72
contents area . . . 12
conversion formats . . . 3
copyright . . . 27
covered/1 . . . 71, 72
covered/2 . . . 71, 72
csh . . . 12, 14, 15, 101

D

Daniel Cabeza . . . 55
data_facts . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
date . . . 23
debugger_support . . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
decl assertion . . . 41
decl/1 . . . 36, 41, 45
decl/2 . . . 36, 41
deprecated/1 . . . 55, 67
description list . . . 22
dict . . . 106, 125, 137, 145
dictionary/1 . . . 45, 48
dir . . . 14, 101
distutils/dirutils . . . 127, 129
distutils/distutils . . . 119, 127, 129, 133
doc/2 . . . 13, 16, 17, 19, 26, 30, 31, 36, 41, 42
doc_customdic_get/3 . . . 106, 109
doc_customdic_lookup/3 . . . 106, 109
doc_customdic_maybe_get/3 . . . 106, 109
doc_customdic_replace/4 . . . 106, 109
doc_doing_mainmod/1 . . . 106, 109
doc_id_type/3 . . . 19, 26
doc_message/2 . . . 106, 109
doc_message/3 . . . 106, 109
doc_modname/2 . . . 106, 109
doc_opt/2 . . . 106, 109
doc_structure/1 . . . 10
doclabel/1 . . . 113, 114
doclink/1 . . . 113, 114, 141
doclink_at/2 . . . 113, 114
doclink_is_local/1 . . . 113, 114
docstate(Arg1) . . . 140
docstate(Arg3) . . . 116
docstate(DocSt) . . 115, 116, 123, 125, 131, 141, 145, 149
docstate/1 . . . 106, 108, 115, 116, 123, 125, 131, 140, 141, 145, 149
docstate_backend/2 . . . 106, 108
docstate_currmod/2 . . . 106, 108
docstate_has_index/2 . . . 106, 109
docstate_inputfile/2 . . . 106, 109
docstate_opts/2 . . . 106, 108
docstate_set_currmod/3 . . . 106, 108
docstate_set_customdic/3 . . . 106, 109
docstate_set_opts/3 . . . 106, 108
docstr_node/4 . . . 117
docstring/1 . . . 19, 26, 27, 28, 29, 30, 35, 42, 45, 46, 48, 49, 50
doctokens/1 . . . 113, 114
doctree/1 . . . 113, 114, 115, 116, 123, 125, 131, 138, 141
doctree_is_empty/1 . . . 113, 114
doctree_prepare_refs_translate_and_write/3 . . . 113, 115
doctree_putvars/5 . . . 113, 115
doctree_restore/2 . . . 113, 115
doctree_save/2 . . . 113, 115
doctree_scan_and_save_refs/2 . . . 113, 115
doctree_simplify/2 . . . 113, 115
doctree_to_rawtext/3 . . . 113, 115
doctree_translate_and_write/3 . . . 113, 116
document structure . . . 10
documentation format . . . 105
documentation strings . . . 26
DOTcshrc . . . 101
dvips . . . 102
dynamic/1 . . . 13

E

Edison Mera 71, 97, 99
emacs 30, 33, 101, 102
Emacs 9, 12, 13, 14
emacs Ciao mode 29, 30
Emacs, accessing info files 13
Emacs, generating manuals from 9
Emacs, LPdoc mode 9
emacs-library.el 102
email address 22
email addresses 21
emphasis face 23
empty_doctree/1 .... 113, 114
encapsulated postscript 25
engine/basic_props 89
ensure_cache_dir/1 .... 133, 134
ensure_loaded/1 .... 16, 31
ensure_output_dir/1 .... 133, 134
ensure_output_dir_prepared/2 .... 106, 110
entry assertion 40
entry/1 .... 36, 40, 48
enumerated list 22
environment variables 101
equiv/2 .... 55, 69
errhandle 149
error(a,b) 99
error_free/1 .... 55, 69
error_text/3 143
escape sequences 20
escape_string/4 .... 113, 116
eval/1 .. 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 69, 93, 95
example of lpdoc use 83
example_module 89
exception(error(a,b)) 99
exception/1 .... 71, 72
exception/2 .... 71, 72
exceptions .. 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
exit assertion 40, 41
exit/1 .... 36, 40, 41
exit/2 .... 36, 41
exported predicates 105
exports 97

F

fails/1 .... 71, 73
false assertion 43
false/1 .... 36, 43
fastformat 123, 125, 131
file_format_name/2 .... 133, 134
file_format_provided_by_backend/3 ..... 133, 134
file_utils 71, 123, 125, 127, 129, 145
filename/1 133
filenames 106, 117
filetype/1 .... 19, 26, 31
filter/2 .... 55, 69
finite_solutions/1 .... 71, 73
Firefox 13
fixed format text 23
fixed-width font 23
flag_values/1 .... 55, 69
flt/1 .... 55, 57
fmt_html_template/3 129
fmt_idx_env/7 .... 137, 138
fmt_index/4 .... 137, 138
foo 89
foo(Arg1) 94
footnote 23
format 106, 113, 123, 145, 149
formatting commands 3, 19, 35
framed box 23
Francisco Bueno 35, 51, 71, 81
func/1 49

G

g_assrt_body/1 .... 39, 45, 49
generating from Emacs 9
generating manuals 9
German Puebla 35
get_autodoc_opts/3 .... 106, 110
get_cache_dir/2 .... 133, 134
get_command_option/1 .... 119, 120
get_doc/4 .... 106, 110
get_idxbase/3 137
get_idxsub/2 137
get_main_title/2 .... 139, 141
get_mainmod/1 .... 117, 118
get_mainmod_spec/1 .... 117, 118
get_output_dir/2 .... 133, 134
get_subbase/3 .... 133, 135

gmake 101
gnd/1 . . . . . . 55, 59, 90, 91, 92, 93, 94, 96
gndstr/1 . . . . . . 55, 59, 60
GNU general public license 1
GNU Make 101
ground/1 . . . 59, 60, 62, 63, 64, 65, 66, 67, 71, 72, 74, 91, 93, 95
gunzip 101

H

hard side-effects 77
have_choicepoints/1 . . . . . . 71, 73
head pattern 45, 46, 49
head_pattern/1 . . . . . . 42, 45, 46, 49
hiord_rt 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
html 4
html index page 12

I

idx_get_indices/3 . . . . . . 137, 138
image file 25
images, inserting 25
images, scaling 25
img_url/2 129
include files 16, 31
include/1 . . . . . . 16, 31
including a predicate definition 25
including an image 25
including code 25
including files 25
including images 25
including or not authors 10
including or not bug info 10
including or not changelog 10
including or not versions, patches 10
indentation, avoiding 23
indep/1 . . . . . . 71, 72, 73, 74
indep/2 . . . . . . 71, 72, 73, 74
index pages out of order 12
index_comment/2 . . . . . . 10, 106, 107
info 1, 4, 13, 14, 15, 18, 23, 101
info path list 14
infodir_base/2 . . . . . . 106, 111
inner 97
insert_printtoc/3 . . . . . . 113, 116
inst/2 . . . . . . 55, 66
installation 3
installation, of manuals 12
INSTALLATION.lpdoc 17
instance/2 80
instantiation properties 51
int/1 . . . . . . 55, 56, 76, 90, 91, 92, 94, 95
integer/1 47
internals 71
internals manual 3, 16
intervals/2 . . . . . . 71, 80
introduction 28
io_aux . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
io_basic 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
is/2 96
is_det/1 . . . . . . 56, 57, 58, 59, 60, 61, 62, 71, 73, 93
is_index_cmd/1 . . . . . . 137, 138
is_list/1 . . . . . . 117, 118
is_nonempty_doctree/1 . . . . . . 113, 114
iso/1 . . . . . . 55, 67, 94
italics face 23
item in an itemized list 22
itemized list 22

J

Jose F. Morales . . . 105, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 145, 147, 149

K

keyboard key 23
known_format/1 149

L

labgen_clean/1 139
labgen_get/2 139
labgen_init/1 139
LaTeX 20
LaTeX notation 23
letter size paper 11
library 3

library(basicmodes) 46
library(isomodes) 46
linear/1 71, 74
Linux 101, 102
list/1 55, 61, 62, 63, 66, 76, 90, 91, 92, 93, 94, 95
list/2 31, 47, 55, 62, 91, 92, 95, 110, 111, 149
list_or_aorb/2 89, 90
lists 36, 39, 71, 89, 106, 113, 123, 125, 129, 131, 137, 139, 145, 147
literal 98
literate programming 13
load_vpaths/0 119, 120
locate_and_convert_image/4 149
log of changes 29
long 98
long/1 89, 91, 92
lpdoc 1, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 22, 30, 35, 42, 46, 50, 83, 101, 102
lpdoc -help 10
lpdoc all 10, 11, 13
lpdoc_option/1 119
lpdocsrc(makedir(LPDOCSETTINGS)) 123, 145, 149
lpdocsrc(src(autodoc)) 113, 123, 125, 127, 131, 133, 137, 139, 145, 149
lpdocsrc(src(autodoc_aux)) 123, 131, 145, 149
lpdocsrc(src(autodoc_bibrefs)) 139
lpdocsrc(src(autodoc_doctree)) 106, 123, 125, 131, 137, 139, 145
lpdocsrc(src(autodoc_filesystem)) 106, 113, 123, 125, 127, 137, 139, 149
lpdocsrc(src(autodoc_html)) 113
lpdocsrc(src(autodoc_html_resources)) 106, 125
lpdocsrc(src(autodoc_html_template)) 125
lpdocsrc(src(autodoc_images)) 123, 125, 131
lpdocsrc(src(autodoc_index)) 106, 113, 123, 125
lpdocsrc(src(autodoc_man)) 113
lpdocsrc(src(autodoc_parse)) 106, 145
lpdocsrc(src(autodoc_refsdb)) 106, 113, 125, 137, 145
lpdocsrc(src(autodoc_settings)) 106, 113, 117, 123, 125, 127, 129, 133, 145, 147, 149
lpdocsrc(src(autodoc_structure)) 106, 113, 123, 125, 133, 137, 139
lpdocsrc(src(autodoc_texinfo)) 113
lpdocsrc(src(comments)) 106, 113, 123, 125, 131
lpdocsrc(src(distpkg_download)) 125

M

machine readable comments 19, 26
main body 28
main file 4
main/0 3, 4
main/1 3, 4
main_absfile_for_subtarget/3 133, 135
main_absfile_in_format/2 133, 134
main_output_name/1 133, 135
make/make_rt 106, 113, 117, 119, 123, 129, 145, 149
make/system_extra 106, 113, 119, 123, 127, 129, 133, 145, 147, 149
Makefile 12, 101
makeinfo 102
man 13, 14
Manuel Hermenegildo 9, 19, 35, 45, 51, 55, 71, 83, 101, 105, 113, 123, 131, 143, 145, 147
master index 4
mattr_global 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
member/2 55, 63, 68
memo/1 55, 69
messages 106, 123, 127, 129, 147, 149
meta_predicate/1 13
meta_props 3, 81
mode 36, 46
modedef/1 36, 41, 46
modtype/1 106, 110
module comment 28
module declaration 105
module/1 105
module/2 105
mshare/1 71, 74
mut_exclusive/1 71, 74
mytype/1 89, 92

N

n_assrt_body/5 48, 49
nabody/1 45, 48
native/1 55, 56, 57, 58, 59, 60, 68, 72, 73, 74, 76, 79, 80
native/2 55, 68, 71, 72, 73, 74, 75
native_props 3, 71
nativeprops.pl 97
new item in description list 22

nlist/1 . . . 63
nlist/2 . . . 55, 62
nnegint/1 . . . 55, 56
no . . . 97, 98
no_choicepoints/1 . . . 71, 74
no_exception/1 . . . 71, 75
no_exception/2 . . . 71, 75
no_rtcheck/1 . . . 55, 68
no_signal/1 . . . 71, 75
no_signal/2 . . . 71, 75
non_det/1 . . . 71, 75
nonground/1 . . . 71, 75
nonvar/1 . . . 56, 57, 58, 59, 60, 61, 64, 91, 92
normalize_index_cmd/3 . . . 137, 138
not_covered/1 . . . 71, 75
not_fails . . . 99
not_fails/1 . . . 71, 75, 91, 92, 93, 94, 95
not_further_inst/1 . . . 48
not_further_inst/2 . . . 55, 67, 91, 92
not_mut_exclusive/1 . . . 71, 76
num/1 . . . 55, 57, 58, 96
num_code/1 . . . 55, 65
num_solutions/2 . . . 71, 76

O

og/2 . . . 95
one-sided printing . . . 10
op/3 . . . 13
operator_specifier/1 . . . 55, 61
operators . . . 113
option_comment/2 . . . 10, 106, 107

P

p/1 . . . 89, 91
p/3 . . . 89, 94
p/5 . . . 89, 91
packages . . . 16, 31
page numbering, changing . . . 10
page size, changing . . . 11
page style, changing . . . 11
paragraph break . . . 23
parametric property . . . 82
parametric regular type abstractions . . . 82
parametric type functor . . . 54
parse_structure/0 . . . 117
parse_structure_/2 . . . 117
parse_structure_list/2 . . . 117, 118
parts in a large document . . . 17
parts in large documents . . . 31
pdftex . . . 102
pe_type/1 . . . 55, 70
Pedro Lopez . . . 51, 71
perl . . . 102
pillow/html . . . 129
planned improvement . . . 29
possibly_fails/1 . . . 71, 76
possibly_nondet/1 . . . 71, 77
pred assertion . . . 36, 37
pred/1 . . . 21, 36, 37, 38, 39, 41, 45, 49
pred/2 . . . 36, 37
predfunctor/1 . . . 45, 50
predicate . . . 98
predicate/n . . . 99
predname/1 . . . 29, 30, 31, 46, 55, 65, 66
prepare_current_refs/1 . . . 139, 140
prepare_mathjax/0 . . . 127
prepare_web_skel/1 . . . 127
program assertions . . . 35
program section . . . 28, 29
program subsection . . . 29
Prolog . . . 3, 10, 16, 101
Prolog source files . . . 3
Prolog, Ciao . . . 9
prolog_flags . . . 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
prop assertion . . . 39, 40
prop/1 . . . 36, 39, 40
prop/2 . . . 36, 40, 81
prop_abs/1 . . . 82
properties . . . 3
properties of computations . . . 51
properties of execution states . . . 51
properties, basic . . . 55
properties, native . . . 71
property . . . 39
property abstraction . . . 82
property compatibility . . . 66
property_conjunction/1 . . . 42, 43, 45, 47
property_starterm/1 . . . 45, 47
propfunctor/1 . . . 45, 50
providing information to the compiler . . . 40, 43
pstogif . . . 102

Q

q/1 89, 93
q/2 89, 90

R

r/1 89, 90
read 106, 113, 139
read_file/2 147
references 21, 102
refs_clean/1 139
refs_closure_clean/1 139, 140
refs_closure_entry/3 139, 140
regtype assertion 54
regtype/1 53, 54, 55, 68
regtype/2 53, 54, 81, 82
regtypes 3, 51
regular type 54
regular type definitions 51
regular type expression 54
regular types 51
relations/2 61, 71, 77
requested_file_formats/1 119, 120
reset_output_dir_db/0 106, 110
resolve_bibliography/1 145
rtchecks 97
rtchecks_asrloc 97
rtchecks_callloc 98
rtchecks_entry 97
rtchecks_exit 97
rtchecks_inline 97
rtchecks_level 97
rtchecks_namefmt 98
rtchecks_predloc 97
rtchecks_rt.pl 97
rtchecks_test 97
rtchecks_trust 97
run-time checks 39
running unit tests 99

S

s/1 89, 92
s_assrt_body/1 38, 39, 40, 41, 45, 48
save_refs/1 139, 140
scribe 20
section 23
section_prop/2 113, 115
section_select_prop/3 113, 115
sections 21
secttree/1 139, 140
secttree_resolve/3 139, 140
sequence/2 55, 63
sequence_or_list/2 55, 64
setting_value/2 119, 120
setting_value_or_default/2 119
setting_value_or_default/3 119
SETTINGS 5, 22
SETTINGS.pl 9, 10, 12, 13, 14, 15, 17
SETTINGS.pl.generated 9
sh_exec/2 147
sharing pieces of text 25
sharing sets 74
short 98
sideff/2 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 93, 95
sideff_hard/1 71, 77
sideff_pure/1 71, 77
sideff_soft/1 71, 77
signal/1 71, 77
signal/2 71, 77
signals/2 71, 78
size/2 71, 78
size/3 71, 78
size_lb/2 71, 78
size_metric/3 71, 78
size_metric/4 71, 78
size_o/2 71, 78
size_ub/2 71, 78
soft side-effects 77
solutions/2 71, 76
sort 71
space, extra lines 23
spcae, horizontal fill 23
special characters 24
specifications 35
standalone_docstr/1 117, 118
steps/2 71, 79
steps_lb/2 71, 79, 91
steps_o/2 71, 79
steps_ub/2 71, 79
streams 71
streams_basic 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149

string/1 ... 55, 65, 107, 108, 116, 129, 138, 141, 149
stringcommand/1 .... 19, 20, 42, 46, 48, 49, 50
strings 19
strong face 23
struct/1 .... 55, 58, 59
subsection 23
subtarget/1 133
subtitle 27
success assertion 38
success/1 .... 36, 38, 40
success/2 .... 36, 38
supported documentation formats 110
supported_file_format/1 .... 133, 134
supported_option/1 .... 106, 107, 108
synopsis section of the man page 10
syntax of formatting commands 20
system 71, 106, 123, 125, 129, 133, 139, 147, 149
system modules 10
system_info .. 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149

T

t/5 .... 89, 92
tar 101
tau/1 .... 71, 79
term/1 .... 55, 63
term_basic ... 19, 36, 45, 53, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
term_compare 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
term_typing .. 19, 36, 45, 55, 71, 81, 89, 98, 100, 106, 113, 117, 119, 123, 125, 127, 129, 131, 133, 137, 139, 143, 145, 147, 149
terminates/1 .... 71, 80
terms 106, 113, 117, 123, 127, 129, 133, 145, 149
terms_check 55, 71
terms_vars 71
test assertion 38, 39, 99
test/1 .... 36, 38, 39
test/2 .... 36, 38
test_type/2 .... 56, 57, 58, 59, 71, 80
tex 18, 102
texec assertion 37
texec/1 .... 36, 37
texec/2 .... 36, 37
texi2dvi 102
texi2html 102
texindex 102
texinfo 1, 3, 4, 10, 11, 15, 18, 105
Texinfo 102
texinfo files 4
textual comments 19
thesis-like style 11
throws/2 .... 71, 80
time_struct/1 32
times(N) 99
title 26
top_suffix/2 .... 106, 107
tree_of/2 .... 89, 90
troubleshooting 9
true assertion 43
true/1 .... 36, 43
trust assertion 42
trust/1 .... 36, 42
try_sols(N) 99
ttyout 106
two-sided 10
typeindex/5 137
types 3
typewriter-like font 23

U

u/3 .... 89, 91
unit tests 99
unittest 99
Universal Resource Locator 22
Unix 14
URL 22
urls 21
usage 36
usage of a command 25
usage of the application 10
usage tips 9
usage_message/1 10
use_module/1 17
use_package/1 .... 16, 31
user_output/2 .... 71, 80
using citations 10
using_mathjax/1 127

V

var/1 . . . 47, 90, 91, 92, 94, 95
variable names . . . 35
verbatim text . . . 23
version maintenance mode for packages . . . 30
version number . . . 29
version_date/2 . . . 106, 111
version_descriptor/1 . . . 19, 25, 30
version_maintenance_type/1 . . . 30, 32
version_number/1 . . . 32
version_numstr/2 . . . 106, 111
version_patch/2 . . . 106, 111

W

w/1 . . . 89, 92
word-help . . . 14
word-help-setup.el . . . 14
word-help.el . . . 14
write . . . 113, 139
WWW . . . 1, 13
WWW address . . . 22

X

xdvi . . . 11

Y

yes . . . 97, 98
ymd_date/1 . . . 32