GNU PROLOG

A Native Prolog Compiler with Constraint Solving over Finite Domains

Edition 1.7, for GNU Prolog version 1.2.16
September 25, 2002

by Daniel Diaz

Contents

1 Acknowledgements **9**

2 Introduction **11**

3 Using GNU Prolog **13**
3.1 Introduction . . . 13
3.2 The GNU Prolog interactive interpreter . . . 13
3.2.1 Starting/exiting the interactive interpreter . . . 13
3.2.2 The interactive interpreter read-execute-write loop . . . 14
3.2.3 Consulting a Prolog program . . . 16
3.2.4 Interrupting a query . . . 17
3.2.5 The line editor . . . 18
3.3 Adjusting the size of Prolog stacks . . . 19
3.4 The GNU Prolog compiler . . . 20
3.4.1 Different kinds of codes . . . 20
3.4.2 Compilation scheme . . . 20
3.4.3 Using the compiler . . . 22
3.4.4 Running an executable . . . 25
3.4.5 Generating a new interactive interpreter . . . 25
3.4.6 The hexadecimal predicate name encoding . . . 26

4 Debugging **29**
4.1 Introduction . . . 29
4.2 The procedure box model . . . 29
4.3 Debugging predicates . . . 29
4.3.1 Running and stopping the debugger . . . 29
4.3.2 Leashing ports . . . 30
4.3.3 Spy-points . . . 30
4.4 Debugging messages . . . 31
4.5 Debugger commands . . . 31
4.6 The WAM debugger . . . 32

5 Format of definitions **35**
5.1 General format . . . 35
5.2 Types and modes . . . 35
5.3 Errors . . . 37
5.3.1 General format and error context . . . 37
5.3.2 Instantiation error . . . 37
5.3.3 Type error . . . 38
5.3.4 Domain error . . . 38
5.3.5 Existence error . . . 39
5.3.6 Permission error . . . 39
5.3.7 Representation error . . . 39
5.3.8 Evaluation error . . . 40
5.3.9 Resource error . . . 40
5.3.10 Syntax error . . . 40
5.3.11 System error . . . 40

6 Prolog directives and control constructs **41**
6.1 Prolog directives . . . 41
6.1.1 Introduction . . . 41
6.1.2 `dynamic/1` . . . 41
6.1.3 `public/1` . . . 41
6.1.4 `multifile/1` . . . 42
6.1.5 `discontiguous/1` . . . 42
6.1.6 `ensure_linked/1` . . . 43

6.1.7 built_in/0, built_in/1, built_in_fd/0, built_in_fd/1 . . . 43
6.1.8 include/1 . . . 44
6.1.9 ensure_loaded/1 . . . 44
6.1.10 op/3 . . . 44
6.1.11 char_conversion/2 . . . 44
6.1.12 set_prolog_flag/2 . . . 45
6.1.13 initialization/1 . . . 45
6.1.14 foreign/2, foreign/1 . . . 45
6.2 Prolog control constructs . . . 46
6.2.1 true/0, fail/0, !/0 . . . 46
6.2.2 (',')/2 - conjunction, (;)/2 - disjunction, (->)/2 - if-then . . . 46
6.2.3 call/1 . . . 47
6.2.4 catch/3, throw/1 . . . 47

7 Prolog built-in predicates **49**
7.1 Type testing . . . 49
7.1.1 var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1, callable/1, list/1, partial_list/1, list_or_partial_list/1 . . 49
7.2 Term unification . . . 50
7.2.1 (=)/2 - Prolog unification . . . 50
7.2.2 unify_with_occurs_check/2 . . . 50
7.2.3 (\=)/2 - not Prolog unifiable . . . 50
7.3 Term comparison . . . 51
7.3.1 Standard total ordering of terms . . . 51
7.3.2 (==)/2 - term identical, (\==)/2 - term not identical, (@<)/2 - term less than, (@=<)/2 - term less than or equal to, (@>)/2 - term greater than, (@>=)/2 - term greater than or equal to . . . 51
7.3.3 compare/3 . . . 52
7.4 Term processing . . . 52
7.4.1 functor/3 . . . 52
7.4.2 arg/3 . . . 53
7.4.3 (=..)/2 - univ . . . 53
7.4.4 copy_term/2 . . . 54
7.4.5 setarg/4, setarg/3 . . . 54
7.5 Variable naming/numbering . . . 55
7.5.1 name_singleton_vars/1 . . . 55
7.5.2 name_query_vars/2 . . . 55
7.5.3 bind_variables/2, numbervars/3, numbervars/1 . . . 56
7.5.4 term_ref/2 . . . 57
7.6 Arithmetic . . . 57
7.6.1 Evaluation of an arithmetic expression . . . 57
7.6.2 (is)/2 - evaluate expression . . . 59
7.6.3 (=:=)/2 - arithmetic equal, (=\=)/2 - arithmetic not equal, (<)/2 - arithmetic less than, (=<)/2 - arithmetic less than or equal to, (>)/2 - arithmetic greater than, (>=)/2 - arithmetic greater than or equal to . . 60
7.7 Dynamic clause management . . . 60
7.7.1 Introduction . . . 60
7.7.2 asserta/1, assertz/1 . . . 61
7.7.3 retract/1 . . . 62
7.7.4 retractall/1 . . . 62
7.7.5 clause/2 . . . 63
7.7.6 abolish/1 . . . 63
7.8 Predicate information . . . 64
7.8.1 current_predicate/1 . . . 64
7.8.2 predicate_property/2 . . . 65
7.9 All solutions . . . 66
7.9.1 Introduction . . . 66

7.9.2 findall/3 . . . 66
7.9.3 bagof/3, setof/3 . . . 66
7.10 Streams . . . 67
7.10.1 Introduction . . . 67
7.10.2 current_input/1 . . . 69
7.10.3 current_output/1 . . . 69
7.10.4 set_input/1 . . . 69
7.10.5 set_output/1 . . . 70
7.10.6 open/4, open/3 . . . 70
7.10.7 close/2, close/1 . . . 72
7.10.8 flush_output/1, flush_output/0 . . . 72
7.10.9 current_stream/1 . . . 73
7.10.10 stream_property/2 . . . 73
7.10.11 at_end_of_stream/1, at_end_of_stream/0 . . . 74
7.10.12 stream_position/2 . . . 74
7.10.13 set_stream_position/2 . . . 75
7.10.14 seek/4 . . . 75
7.10.15 character_count/2 . . . 76
7.10.16 line_count/2 . . . 77
7.10.17 line_position/2 . . . 77
7.10.18 stream_line_column/3 . . . 77
7.10.19 set_stream_line_column/3 . . . 78
7.10.20 add_stream_alias/2 . . . 79
7.10.21 current_alias/2 . . . 79
7.10.22 add_stream_mirror/2 . . . 79
7.10.23 remove_stream_mirror/2 . . . 80
7.10.24 current_mirror/2 . . . 80
7.10.25 set_stream_type/2 . . . 81
7.10.26 set_stream_eof_action/2 . . . 81
7.10.27 set_stream_buffering/2 . . . 82
7.11 Constant term streams . . . 82
7.11.1 Introduction . . . 82
7.11.2 open_input_atom_stream/2, open_input_chars_stream/2, open_input_codes_stream/2 . . . 83
7.11.3 close_input_atom_stream/1, close_input_chars_stream/1, close_input_codes_stream/1 . . . 83
7.11.4 open_output_atom_stream/1, open_output_chars_stream/1, open_output_codes_stream/1 . . . 84
7.11.5 close_output_atom_stream/2, close_output_chars_stream/2, close_output_codes_stream/2 . . . 84
7.12 Character input/output . . . 85
7.12.1 get_char/2, get_char/1, get_code/1, get_code/2 . . . 85
7.12.2 get_key/2, get_key/1 get_key_no_echo/2, get_key_no_echo/1 . . . 86
7.12.3 peek_char/2, peek_char/1, peek_code/1, peek_code/2 . . . 87
7.12.4 unget_char/2, unget_char/1, unget_code/2, unget_code/1 . . . 88
7.12.5 put_char/2, put_char/1, put_code/1, put_code/2, nl/1, nl/0 . . . 88
7.13 Byte input/output . . . 89
7.13.1 get_byte/2, get_byte/1 . . . 89
7.13.2 peek_byte/2, peek_byte/1 . . . 90
7.13.3 unget_byte/2, unget_byte/1 . . . 90
7.13.4 put_byte/2, put_byte/1 . . . 91
7.14 Term input/output . . . 91
7.14.1 read_term/3, read_term/2, read/2, read/1 . . . 92
7.14.2 read_atom/2, read_atom/1, read_integer/2, read_integer/1, read_number/2, read_number/1 . . . 93
7.14.3 read_token/2, read_token/1 . . . 94
7.14.4 syntax_error_info/4 . . . 95

7.14.5 last_read_start_line_column/2 . . . 95
7.14.6 write_term/3, write_term/2, write/2, write/1, writeq/2, writeq/1, write_canonical/2, write_canonical/1, display/2, display/1, print/2, print/1 . . . 96
7.14.7 format/3, format/2 . . . 98
7.14.8 portray_clause/2, portray_clause/1 . . . 99
7.14.9 get_print_stream/1 . . . 100
7.14.10 op/3 . . . 100
7.14.11 current_op/3 . . . 102
7.14.12 char_conversion/2 . . . 102
7.14.13 current_char_conversion/2 . . . 103
7.15 Input/output from/to constant terms . . . 104
7.15.1 read_term_from_atom/3, read_from_atom/2, read_token_from_atom/2 . . . 104
7.15.2 read_term_from_chars/3, read_from_chars/2, read_token_from_chars/2 . . . 104
7.15.3 read_term_from_codes/3, read_from_codes/2, read_token_from_codes/2 . . . 105
7.15.4 write_term_to_atom/3, write_to_atom/2, writeq_to_atom/2, write_canonical_to_atom/2, display_to_atom/2, print_to_atom/2, format_to_atom/3 . . . 105
7.15.5 write_term_to_chars/3, write_to_chars/2, writeq_to_chars/2, write_canonical_to_chars/2, display_to_chars/2, print_to_chars/2, format_to_chars/3 . . . 106
7.15.6 write_term_to_codes/3, write_to_codes/2, writeq_to_codes/2, write_canonical_to_codes/2, display_to_codes/2, print_to_codes/2, format_to_codes/3 . . . 106
7.16 DEC-10 compatibility input/output . . . 107
7.16.1 Introduction . . . 107
7.16.2 see/1, tell/1, append/1 . . . 107
7.16.3 seeing/1, telling/1 . . . 108
7.16.4 seen/0, told/0 . . . 108
7.16.5 get0/1, get/1, skip/1 . . . 108
7.16.6 put/1, tab/1 . . . 109
7.17 Term expansion . . . 109
7.17.1 Definite clause grammars . . . 109
7.17.2 expand_term/2, term_expansion/2 . . . 111
7.17.3 phrase/3, phrase/2 . . . 111
7.18 Logic, control and exceptions . . . 112
7.18.1 abort/0, stop/0, top_level/0, break/0, halt/1, halt/0 . . . 112
7.18.2 once/1, (\+)/1 - not provable, call_with_args/1-11, call/2 . . . 113
7.18.3 repeat/0 . . . 113
7.18.4 for/3 . . . 114
7.19 Atomic term processing . . . 114
7.19.1 atom_length/2 . . . 114
7.19.2 atom_concat/3 . . . 115
7.19.3 sub_atom/5 . . . 115
7.19.4 char_code/2 . . . 116
7.19.5 lower_upper/2 . . . 116
7.19.6 atom_chars/2, atom_codes/2 . . . 116
7.19.7 number_atom/2, number_chars/2, number_codes/2 . . . 117
7.19.8 name/2 . . . 118
7.19.9 atom_hash/2 . . . 119
7.19.10 new_atom/3, new_atom/2, new_atom/1 . . . 119
7.19.11 current_atom/1 . . . 120
7.19.12 atom_property/2 . . . 120
7.20 List processing . . . 121
7.20.1 append/3 . . . 121
7.20.2 member/2, memberchk/2 . . . 121
7.20.3 reverse/2 . . . 122

7.20.4 delete/3, select/3 . . . 122
7.20.5 permutation/2 . . . 122
7.20.6 prefix/2, suffix/2 . . . 123
7.20.7 sublist/2 . . . 123
7.20.8 last/2 . . . 124
7.20.9 length/2 . . . 124
7.20.10 nth/3 . . . 124
7.20.11 max_list/2, min_list/2, sum_list/2 . . . 125
7.20.12 sort/2, sort0/2, keysort/2 sort/1, sort0/1, keysort/1 . . . 125
7.21 Global variables . . . 126
7.21.1 Introduction . . . 126
7.21.2 g_assign/2, g_assignb/2, g_link/2 . . . 127
7.21.3 g_read/2 . . . 128
7.21.4 g_array_size/2 . . . 128
7.21.5 g_inc/3, g_inc/2, g_inco/2, g_inc/1, g_dec/3, g_dec/2, g_deco/2, g_dec/1 . . . 129
7.21.6 g_set_bit/2, g_reset_bit/2, g_test_set_bit/2, g_test_reset_bit/2 . . . 130
7.21.7 Examples . . . 130
7.22 Prolog state . . . 133
7.22.1 set_prolog_flag/2 . . . 133
7.22.2 current_prolog_flag/2 . . . 134
7.22.3 set_bip_name/2 . . . 135
7.22.4 current_bip_name/2 . . . 135
7.22.5 write_pl_state_file/1, read_pl_state_file/1 . . . 136
7.23 Program state . . . 136
7.23.1 consult/1, '.'/2 - program consult . . . 136
7.23.2 load/1 . . . 137
7.23.3 listing/1, listing/0 . . . 138
7.24 System statistics . . . 138
7.24.1 statistics/0, statistics/2 . . . 138
7.24.2 user_time/1, system_time/1, cpu_time/1, real_time/1 . . . 139
7.25 Random number generator . . . 139
7.25.1 set_seed/1, randomize/0 . . . 139
7.25.2 get_seed/1 . . . 140
7.25.3 random/1 . . . 140
7.25.4 random/3 . . . 140
7.26 File name processing . . . 141
7.26.1 absolute_file_name/2 . . . 141
7.26.2 decompose_file_name/4 . . . 141
7.26.3 prolog_file_name/2 . . . 142
7.27 Operating system interface . . . 142
7.27.1 argument_counter/1 . . . 142
7.27.2 argument_value/2 . . . 143
7.27.3 argument_list/1 . . . 143
7.27.4 environ/2 . . . 144
7.27.5 make_directory/1, delete_directory/1, change_directory/1 . . . 144
7.27.6 working_directory/1 . . . 144
7.27.7 directory_files/2 . . . 145
7.27.8 rename_file/2 . . . 145
7.27.9 delete_file/1, unlink/1 . . . 146
7.27.10 file_permission/2, file_exists/1 . . . 146
7.27.11 file_property/2 . . . 147
7.27.12 temporary_name/2 . . . 148
7.27.13 temporary_file/3 . . . 149
7.27.14 date_time/1 . . . 149
7.27.15 host_name/1 . . . 150
7.27.16 os_version/1 . . . 150
7.27.17 architecture/1 . . . 151

7.27.18 shell/2, shell/1, shell/0 . . . 151
7.27.19 system/2, system/1 . . . 152
7.27.20 spawn/3, spawn/2 . . . 152
7.27.21 popen/3 . . . 153
7.27.22 exec/5, exec/4 . . . 153
7.27.23 fork_prolog/1 . . . 154
7.27.24 create_pipe/2 . . . 154
7.27.25 wait/2 . . . 155
7.27.26 prolog_pid/1 . . . 155
7.27.27 send_signal/2 . . . 156
7.27.28 sleep/1 . . . 156
7.27.29 select/5 . . . 156
7.28 Sockets input/output . . . 157
7.28.1 Introduction . . . 157
7.28.2 socket/2 . . . 158
7.28.3 socket_close/1 . . . 158
7.28.4 socket_bind/2 . . . 159
7.28.5 socket_connect/4 . . . 159
7.28.6 socket_listen/2 . . . 160
7.28.7 socket_accept/4, socket_accept/3 . . . 160
7.28.8 hostname_address/2 . . . 161
7.29 Linedit management . . . 162
7.29.1 get_linedit_prompt/1 . . . 162
7.29.2 set_linedit_prompt/1 . . . 162
7.29.3 add_linedit_completion/1 . . . 162
7.29.4 find_linedit_completion/2 . . . 163
7.30 Source reader facility . . . 163
7.30.1 Introduction . . . 163
7.30.2 sr_open/3 . . . 164
7.30.3 sr_change_options/2 . . . 164
7.30.4 sr_close/1 . . . 164
7.30.5 sr_read_term/4 . . . 164
7.30.6 sr_current_descriptor/1 . . . 164
7.30.7 sr_get_stream/2 . . . 164
7.30.8 sr_get_module/3 . . . 164
7.30.9 sr_get_file_name/2 . . . 164
7.30.10 sr_get_position/3 . . . 164
7.30.11 sr_get_include_list/2 . . . 164
7.30.12 sr_get_include_stream_list/2 . . . 164
7.30.13 sr_get_size_counters/3 . . . 164
7.30.14 sr_get_error_counters/3 . . . 164
7.30.15 sr_set_error_counters/3 . . . 164
7.30.16 sr_error_from_exception/2 . . . 164
7.30.17 sr_write_message/8, sr_write_message/6, sr_write_message/4 . . . 164
7.30.18 sr_write_error/6, sr_write_error/4, sr_write_error/2 . . . 164

8 Finite domain solver and built-in predicates 165
8.1 Introduction . . . 165
8.1.1 Finite Domain variables . . . 165
8.2 FD variable parameters . . . 166
8.2.1 fd_max_integer/1 . . . 166
8.2.2 fd_vector_max/1 . . . 166
8.2.3 fd_set_vector_max/1 . . . 167
8.3 Initial value constraints . . . 167
8.3.1 fd_domain/3, fd_domain_bool/1 . . . 167
8.3.2 fd_domain/2 . . . 168
8.4 Type testing . . . 168

8.4.1 fd_var/1, non_fd_var/1, generic_var/1, non_generic_var/1 168
8.5 FD variable information 169
8.5.1 fd_min/2, fd_max/2, fd_size/2, fd_dom/2 169
8.5.2 fd_has_extra_cstr/1, fd_has_vector/1, fd_use_vector/1 170
8.6 Arithmetic constraints 170
8.6.1 FD arithmetic expressions 170
8.6.2 Partial AC: (#=)/2 - constraint equal, (#\=)/2 - constraint not equal, (#<)/2 - constraint less than, (#=<)/2 - constraint less than or equal, (#>)/2 - constraint greater than, (#>=)/2 - constraint greater than or equal . . . 171
8.6.3 Full AC: (#=#)/2 - constraint equal, (#\=#)/2 - constraint not equal, (#<#)/2 - constraint less than, (#=<#)/2 - constraint less than or equal, (#>#)/2 - constraint greater than, (#>=#)/2 - constraint greater than or equal . . 172
8.6.4 fd_prime/1, fd_not_prime/1 173
8.7 Boolean and reified constraints 173
8.7.1 Boolean FD expressions 173
8.7.2 (#\)/1 - constraint NOT, (#<=>)/2 - constraint equivalent, (#\<=>)/2 - constraint different, (##)/2 - constraint XOR, (#==>)/2 - constraint imply, (#\==>)/2 - constraint not imply, (#/\)/2 - constraint AND, (#\/\)/2 - constraint NAND, (#\/)/2 - constraint OR, (#\\/)/2 - constraint NOR 174
8.7.3 fd_cardinality/2, fd_cardinality/3, fd_at_least_one/1, fd_at_most_one/1, fd_only_one/1 175
8.8 Symbolic constraints 176
8.8.1 fd_all_different/1 176
8.8.2 fd_element/3 176
8.8.3 fd_element_var/3 177
8.8.4 fd_atmost/3, fd_atleast/3, fd_exactly/3 177
8.8.5 fd_relation/2, fd_relationc/2 178
8.9 Labeling constraints 179
8.9.1 fd_labeling/2, fd_labeling/1, fd_labelingff/1 179
8.10 Optimization constraints 180
8.10.1 fd_minimize/2, fd_maximize/2 180

9 Interfacing Prolog and C 183
9.1 Calling C from Prolog 183
9.1.1 Introduction 183
9.1.2 foreign/2 directive 183
9.1.3 The C function 184
9.1.4 Input arguments 185
9.1.5 Output arguments 185
9.1.6 Input/output arguments 185
9.1.7 Writing non-deterministic C code 186
9.1.8 Example: input and output arguments 186
9.1.9 Example: non-deterministic code 187
9.1.10 Example: input/output arguments 189
9.2 Manipulating Prolog terms 190
9.2.1 Introduction 190
9.2.2 Managing Prolog atoms 191
9.2.3 Reading Prolog terms 191
9.2.4 Unifying Prolog terms 192
9.2.5 Creating Prolog terms 194
9.2.6 Testing the type of Prolog terms 194
9.2.7 Comparing Prolog terms 195
9.2.8 Copying Prolog terms 195
9.2.9 Comparing and evaluating arithmetic expressions 196
9.3 Raising Prolog errors 196
9.3.1 Managing the error context 196

9.3.2 Instantiation error 196
9.3.3 Type error 196
9.3.4 Domain error 197
9.3.5 Existence error 197
9.3.6 Permission error 197
9.3.7 Representation error 197
9.3.8 Evaluation error 198
9.3.9 Resource error 198
9.3.10 Syntax error 198
9.3.11 System error 198
9.4 Calling Prolog from C 199
9.4.1 Introduction 199
9.4.2 Example: `my_call/1` - a `call/1` clone 200
9.4.3 Example: recovering the list of all operators 202
9.5 Defining a new C `main()` function 203
9.5.1 Example: asking for ancestors 204

References **207**

Index **209**

1 Acknowledgements

I would like to thank the department of computing science[2] at the university of Paris 1 for allowing me the time and freedom necessary to achieve this project.

I am grateful to the members of the Loco project[3] at INRIA Rocquencourt[4] for their encouragement. Their involvement in this work led to useful feedback and exchange.

I would particularly like to thank Jonathan Hodgson[5] for the time and effort he put into the proofreading of this manual. His suggestions, both regarding ISO technical aspects as well as the language in which it was expressed, proved invaluable.

The on-line HTML version of this document was created using HEVEA[6] developed by Luc Maranget who kindly devoted so much of his time extending the capabilities of HEVEA in order to handle such a sizeable manual.

Jean-Christophe Aude kindly improved the visual aspect of both the illustrations and the GNU Prolog web pages.

Thanks to Richard A. O'Keefe for his advice regarding the implementation of some Prolog built-in predicates and for suggesting me the in-place installation feature.

Many thanks to the following contributors:

- Alexander Diemand[7] for his initial port to alpha/linux and more generally for his personal involvement in the development of GNU Prolog.
- Clive Cox[8] and Edmund Grimley Evans[9] for their port to ix86/SCO.
- Nicolas Ollinger[10] to for his port to ix86/FreeBSD.
- Brook Milligan[11] for his port to ix86/NetBSD and for general configuration improvements.
- Andreas Stolcke[12] for his port to ix86/Solaris.
- Lindsey Spratt[13] for his port to powerpc/Darwin (MacOS X).

Many thanks to all those people at GNU[14] who helped me to finalize the GNU Prolog project.

Finally, I would like to thank everybody who tested preliminary releases and helped me to put the finishing touches to this system.

[2] `http://panoramix.univ-paris1.fr/CRINFO/`
[3] `http://loco.inria.fr/`
[4] `http://www.inria.fr/Unites/ROCQUENCOURT-eng.html`
[5] `http://www.sju.edu/~jhodgson`
[6] `http://pauillac.inria.fr/~maranget/hevea/`
[7] `ax@apax.net`
[8] `clive@laluna.demon.co.uk`
[9] `http://www.rano.org/`
[10] `nollinge@ens-lyon.fr`
[11] `brook@nmsu.edu`
[12] `http://www.speech.sri.com/people/stolcke/`
[13] `spratt@alum.mit.edu`
[14] `http://www.gnu.org`

2 Introduction

GNU Prolog is a free Prolog compiler with constraint solving over finite domains developed by Daniel Diaz[15]. For recent information about GNU Prolog please consult the GNU Prolog page[16].

GNU Prolog is a Prolog compiler based on the Warren Abstract Machine (WAM) [8, 1]. It first compiles a Prolog program to a WAM file which is then translated to a low-level machine independent language called mini-assembly specifically designed for GNU Prolog. The resulting file is then translated to the assembly language of the target machine (from which an object is obtained). This allows GNU Prolog to produce a native stand alone executable from a Prolog source (similarly to what does a C compiler from a C program). The main advantage of this compilation scheme is to produce native code and to be fast. Another interesting feature is that executables are small. Indeed, the code of most unused built-in predicates is not included in the executables at link-time.

A lot of work has been devoted to the ISO compatibility. Indeed, GNU Prolog is very close to the ISO standard for Prolog[17] [5].

GNU Prolog also offers various extensions very useful in practice (global variables, OS interface, sockets,...). In particular, GNU Prolog contains an efficient constraint solver over Finite Domains (FD). This opens contraint logic pogramming to the user combining the power of constraint programming to the declarativity of logic programming. The key feature of the GNU Prolog solver is the use of a single (low-level) primitive to define all (high-level) FD constraints. There are many advantages of this approach: constraints can be compiled, the user can define his own constraints (in terms of the primitive), the solver is open and extensible (as opposed to black-box solvers like CHIP),... Moreover, the GNU Prolog solver is rather efficient, often more than commercial solvers.

GNU Prolog is inspired from two systems developed by the same author:

- `wamcc`: a Prolog to C compiler [3]. the key point of `wamcc` was its ability to produce stand alone executables using an original compilation scheme: the translation of Prolog to C via the WAM. Its drawback was the time needed by `gcc` to compile the produced sources. GNU Prolog can also produce stand alone executables but using a faster compilation scheme.
- `clp(FD)`: a constraint programming language over FD [4]. Its key feature was the use of a single primitive to define FD constraints. GNU Prolog is based on the same idea but offers an extended constraint definition language. In comparison to `clp(FD)`, GNU Prolog offers new predefined constraints, new predefined heuristics, reified constraints,...

Here are some features of GNU Prolog:

- Prolog system:
  - conforms to the ISO standard for Prolog (floating point numbers, streams, dynamic code,...).
  - a lot of extensions: global variables, definite clause grammars (DCG), sockets interface, operating system interface,...
  - more than 300 Prolog built-in predicates.
  - Prolog debugger and a low-level WAM debugger.
  - line editing facility under the interactive interpreter with completion on atoms.
  - powerful bidirectional interface between Prolog and C.
- Compiler:
  - native-code compiler producing stand alone executables.
  - simple command-line compiler accepting a wide variety of files: Prolog files, C files, WAM files,...

[15] `http://pauillac.inria.fr/~diaz`
[16] `http://www.gnu.org/software/prolog`
[17] `http://www.logic-programming.org/prolog_std.html`

  - direct generation of assembly code 15 times faster than `wamcc` + `gcc`.
  - most of unused built-in predicates are not linked (to reduce the size of the executables).
  - compiled predicates (native-code) as fast as `wamcmcc` on average.
  - consulted predicates (byte-code) 5 times faster than `wamcc`.

- Constraint solver:
  - FD variables well integrated into the Prolog environment (full compatibility with Prolog variables and integers). No need for explicit FD declarations.
  - very efficient FD solver (comparable to commercial solvers).
  - high-level constraints can be described in terms of simple primitives.
  - a lot of predefined constraints: arithmetic constraints, boolean constraints, symbolic constraints, reified constraints,...
  - several predefined enumeration heuristics.
  - the user can define his own new constraints.
  - more than 50 FD built-in constraints/predicates.

3 Using GNU Prolog

3.1 Introduction

GNU Prolog offers two ways to execute a Prolog program:

- interpreting it using the GNU Prolog interactive interpreter.
- compiling it to a (machine-dependent) executable using the GNU Prolog native-code compiler.

Running a program under the interactive interpreter allows the user to list it and to make full use of the debugger on it (section 4, page 29). Compiling a program to native code makes it possible to obtain a stand alone executable, with a reduced size and optimized for speed. Running a Prolog program compiled to native-code is around 3-5 times faster than running it under the interpreter. However, it is not possible to make full use of the debugger on a program compiled to native-code. Nor is it possible to list the program. In general, it is preferable to run a program under the interpreter for debugging and then use the native-code compiler to produce an autonomous executable. It is also possible to combine these two modes by producing an executable that contains some parts of the program (e.g. already debugged predicates whose execution-time speed is crucial) and interpreting the other parts under this executable. In that case, the executable has the same facilities as the GNU Prolog interpreter but also integrates the native-code predicates. This way to define a new enriched interpreter is detailed later (section 3.4.5, page 25).

3.2 The GNU Prolog interactive interpreter

3.2.1 Starting/exiting the interactive interpreter

GNU Prolog offers a classical Prolog interactive interpreter also called *top-level*. It allows the user to execute queries, to consult Prolog programs, to list them, to execute them and to debug them. The top-level can be invoked using the following command:

`% gprolog [`*`OPTION`*`]...` (the `%` symbol is the operating system shell prompt)

Options:

`--init-goal` *`GOAL`*	execute *`GOAL`* before top_level/0
`--entry-goal` *`GOAL`*	execute *`GOAL`* inside top_level/0
`--query-goal` *`GOAL`*	execute *`GOAL`* as a query for top_level/0
`--help`	print a help and exit
`--version`	print version number and exit
`--`	do not parse the rest of the command-line

The main role of the `gprolog` command is to execute the top-level itself, i.e. to execute the built-in predicate `top_level/0` (section 7.18.1, page 112) which will produce something like:

```
GNU Prolog 1.2.9
By Daniel Diaz
Copyright (C) 1999-2001 Daniel Diaz
| ?-
```

The top-level is ready to execute your queries as explained in the next section.

To quit the top-level type the end-of-file key sequence (`Ctl-D`) or its term representation: `end_of_file.` It is also possible to use the built-in predicate `halt/0` (section 7.18.1, page 112).

However, before entering the top-level itself, the command-line is processed to treat all known options (those listed above). All unrecognized arguments are collected together to form the argument list which will be available using `argument_value/2` (section 7.27.2, page 143) or `argument_list/1` (section 7.27.3, page 143). The `--` option stops the parsing of the command-line, all remainding options are collected into the argument list.

Several options are provided to execute a goal before entering the interaction with the user:

- The `--init-goal` option executes the *GOAL* as soon as it is encountered (while the commnad-line is processed). *GOAL* is thus executed before entering `top_level/0`.
- The `--entry-goal` option executes the *GOAL* at the entry of `top_level/0` just after the banner is displayed.
- The `--query-goal` option executes the *GOAL* as if the user has typed in.

The above order is thus the order in which each kind of goal (init, entry, query) is executed. If there are several goals of a same kind they are executed in the oder of appearance. Thus, all init goals are executed (in the order of appearance) before all entry goals and all entry goals are executed before all query goals.

Each *GOAL* is passed as a shell argument (i.e. one shell string) and should not contain a terminal dot. Example: `--init-goal 'write(hello), nl'` under a sh-like. To be executed, a *GOAL* is transformed into a term using `read_term_from_atom(Goal, Term, [end_of_term(eof)])`. Respecting both the syntax of shell strings and of Prolog can be heavy. For instance, passing a backslash character \ can be difficult since it introduces an escape sequence both in sh and inside Prolog quoted atoms. The use of back quotes can then be useful since, by default, no escape sequence is processed inside back quotes (this behavior can be controlled using the `back_quotes` Prolog flag (section 7.22.1, page 133)).

Since the Prolog argument list is created when the whole command-line is parsed, if a `--init-goal` option uses `argument_value/2` or `argument_list/1` it will obtained the original command-line arguments (i.e. including all recognized arguments).

Here is an example of using execution goal options:

```
% gprolog --init-goal 'write(before), nl' --entry-goal 'write(inside), nl'
--query-goal 'append([a,b],[c,d],X)'
```

will produce the following:

```
before
GNU Prolog 1.2.9
By Daniel Diaz
Copyright (C) 1999-2001 Daniel Diaz
inside
| ?- append([a,b],[c,d],X).

X = [a,b,c,d]

yes
| ?-
```

3.2.2 The interactive interpreter read-execute-write loop

The GNU Prolog top-level is built on a classical read-execute-write loop that also allows for re-executions (when the query is not deterministic) as follows:

- display the prompt, i.e. '`| ?-`'.
- read a query (i.e. a goal).
- execute the query.

- in case of success display the values of the variables of the query.
- if there are remaining alternatives (i.e. the query is not deterministic), display a `?` and ask the user who can use one of the following commands: `RETURN` to stop the execution, `;` to compute the next solution or `a` to compute all remaining solution.

Here is an example of execution of a query ("find the lists `X` and `Y` such that the concatenation of `X` and `Y` is `[a,b]`"):

```
| ?- append(X,Y,[a,b,c]).

X = []
Y = [a,b,c] ?  ;       (here the user presses ; to compute another solution)

X = [a]
Y = [b,c] ?  a         (here the user presses a to compute all remaining solutions)

X = [a,b]
Y = [c]                (here the user is not asked and the next solution is computed)

X = [a,b,c]
Y = []                 (here the user is not asked and the next solution is computed)

no                     (no more solution)
```

In some cases the top-level can detect that the current solution is the last one (no more alternatives remaining). In such a case it does not display the `?` symbol (and does not ask the user). Example:

```
| ?- (X=1 ; X=2).

X = 1 ?  ;             (here the user presses ; to compute another solution)

X = 2                  (here the user is not prompted since there are no more alternatives)

yes
```

The user can stop the execution even if there are more alternatives by typing `RETURN`.

```
| ?- (X=1 ; X=2).

X = 1 ?                (here the user presses RETURN to stop the execution)

yes
```

The top-level tries to display the values of the variables of the query in a readable manner. For instance, when a variable is bound to a query variable, the name of this variable appears. When a variable is a singleton an underscore symbol `_` is displayed (`_` is a generic name for a singleton variable, it is also called an anonymous variable). Other variables are bound to new brand variable names. When a query variable name `X` appears as the value of another query variable `Y` it is because `X` is itself not instantiated otherwise the value of `X` is displayed. In such a case, nothing is output for `X` itself (since it is a variable). Example:

```
| ?- X=f(A,B,_,A), A=k.

A = k                  (the value of A is displayed also in f/3 for X)
X = f(k,B,_,k)         (since B is a variable which is also a part of X, B is not displayed)

| ?- functor(T,f,3), arg(1,T,X), arg(3,T,X).

T = f(X,_,X)           (the 1st and 3rd args are equal to X, the 2nd is an anonymous variable)

| ?- read_from_atom('k(X,Y,X).',T).

T = k(A,_,A)           (the 1st and 3rd args are unified, a new variable name A is introduced)
```

The top-level uses variable binding predicates (section 7.5, page 55). To display the value of a variable, the top-level calls `write_term/3` with the following option list: `[quoted(true),numbervars(false), namevars(true)]` (section 7.14.6, page 96). A term of the form `'$VARNAME'(Name)` where `Name` is an atom is displayed as a variable name while a term of the form `'$VAR'(N)` where `N` is an integer is displayed as a normal compound term (such a term could be output as a variable name by `write_term/3`). Example:

```
| ?- X='$VARNAME'('Y'), Y='$VAR'(1).

X = Y                   (the term '$VARNAME'('Y') is displayed as Y)
Y = '$VAR'(1)           (the term '$VAR'(1) is displayed as is)

| ?- X=Y, Y='$VAR'(1).

X = '$VAR'(1)
Y = '$VAR'(1)
```

In the first example, `X` is explicitly bound to `'$VARNAME'('Y')` by the query so the top-level displays `Y` as the value of `X`. `Y` is unified with `'$VAR'(1)` so the top-level displays it as a normal compound term. It should be clear that `X` is not bound to `Y` (whereas it is in the second query). This behavior should be kept in mind when doing variable binding operations.

Finally, the top-level computes the user-time (section 7.24.2, page 139) taken by a query and displays it when it is significant. Example:

```
| ?- retractall(p(_)), assertz(p(0)),
     repeat,
        retract(p(X)),
        Y is X + 1,
        assertz(p(Y)),
        X = 1000, !.

X = 1000
Y = 1001

(180 ms) yes            (the query took 180ms of user time)
```

3.2.3 Consulting a Prolog program

The top-level allows the user to consult Prolog source files. Consulted predicates can be listed, executed and debugged (while predicates compiled to native-code cannot). For more information about the difference between a native-code predicate and a consulted predicate refer to the introduction of this section (section 3.1, page 13) and to the part devoted to the compiler (section 3.4.1, page 20).

To consult a program use the built-in predicate `consult/1` (section 7.23.1, page 136). The argument of this predicate is a Prolog file name or `user` to specify the terminal. This allows the user to directly input the predicates from the terminal. In that case the input shall be terminated by the end-of-file key sequence (`Ctl-D`) or its term representation: `end_of_file`. A shorthand for `consult(`*`FILE`*`)` is `[`*`FILE`*`]`. Example:

```
| ?- [user].
{compiling user for byte code...}
even(0).
even(s(s(X))):-
        even(X).
                        (here the user presses Ctl-D to end the input)
{user compiled, 3 lines read - 350 bytes written, 1180 ms}

| ?- even(X).

X = 0 ?  ;              (here the user presses ; to compute another solution)

X = s(s(0)) ?  ;        (here the user presses ; to compute another solution)

X = s(s(s(s(0)))) ?     (here the user presses RETURN to stop the execution)

yes
| ?- listing.

even(0).
even(s(s(A))) :-
        even(A).
```

When `consult/1` (section 7.23.1, page 136) is invoked on a Prolog file it first runs the GNU Prolog compiler (section 3.4, page 20) as a child process to generate a temporary WAM file for byte-code. If the compilation fails a message is displayed and nothing is loaded. If the compilation succeeds, the produced file is loaded into memory using `load/1` (section 7.23.2, page 137). Namely, the byte-code of each predicate is loaded. When a predicate *P* is loaded if there is a previous definition for *P* it is removed (i.e. all clauses defining *P* are erased). We say that *P* is redefined. Note that only consulted predicates can be redefined. If *P* is a native-code predicate, trying to redefine it will produce an error at load-time: the predicate redefinition will be ignored and the following message displayed:

```
native code procedure P cannot be redefined
```

Finally, an existing predicate will not be removed if it is not re-loaded. This means that if a predicate *P* is loaded when consulting the file *F*, and if later the definition of *P* is removed from the file *F*, consulting *F* again will not remove the previously loaded definition of *P* from the memory.

Consulted predicates can be debugged using the Prolog debugger. Use the debugger predicate `trace/0` or `debug/0` (section 4.3.1, page 29) to activate the debugger.

3.2.4 Interrupting a query

Under the top-level it is possible to interrupt the execution of a query by typing the interruption key (`Ctl-C`). This can be used to abort a query, to stop an infinite loop, to activate the debugger,... When an interruption occurs the top-level displays the following message: `Prolog interruption (h for help) ?` The user can then type one of the following commands:

Command	Name	Description
`a`	abort	abort the current execution. Same as `abort/0` (section 7.18.1, page 112)
`e`	exit	quit the current Prolog process. Same as `halt/0` (section 7.18.1, page 112)
`b`	break	invoke a recursive top-level. Same as `break/0` (section 7.18.1, page 112)
`c`	continue	resume the execution
`t`	trace	start the debugger using `trace/0` (section 4.3.1, page 29)
`d`	debug	start the debugger using `debug/0` (section 4.3.1, page 29)
`h` or `?`	help	display a summary of available commands

3.2.5 The line editor

The line editor (`linedit`) allows the user to build/update the current input line using a variety of commands. This facility is available if the `linedit` part of GNU Prolog has been installed. `linedit` is implicitly called by any built-in predicate reading from a terminal (e.g. `get_char/1`, `read/1`,...). This is the case when the top-level reads a query.

Bindings: each command of `linedit` is activated using a key. For some commands another key is also available to invoke the command (on some terminals this other key may not work properly while the primary key always works). Here is the list of available commands:

Key	Alternate key	Description
`Ctl-B`	←	go to the previous character
`Ctl-F`	→	go to the next character
`Esc-B`	`Ctl-`←	go to the previous word
`Esc-F`	`Ctl-`→	go to the next word
`Ctl-A`	`Home`	go to the beginning of the line
`Ctl-E`	`End`	go to the end of the line
`Ctl-H`	`Backspace`	delete the previous character
`Ctl-D`	`Delete`	delete the current character
`Ctl-U`	`Ctl-Home`	delete from beginning of the line to the current character
`Ctl-K`	`Ctl-End`	delete from the current character to the end of the line
`Esc-L`		lower case the next word
`Esc-U`		upper case the next word
`Esc-C`		capitalize the next word
`Ctl-T`		exchange last two characters
`Ctl-V`	`Insert`	switch on/off the insert/replace mode
`Ctl-I`	`Tab`	complete word (twice displays all possible completions)
`Esc-Ctl-I`	`Esc-Tab`	insert spaces to emulate a tabulation
`Ctl-space`		mark beginning of the selection
`Esc-W`		copy (from the begin selection mark to the current character)
`Ctl-W`		cut (from the begin selection mark to the current character)
`Ctl-Y`		paste
`Ctl-P`	↑	recall previous history line
`Ctl-N`	↓	recall next history line
`Esc-P`		recall previous history line beginning with the current prefix
`Esc-N`		recall next history line beginning with the current prefix
`Esc-<`	`Page Up`	recall first history line
`Esc->`	`Page Down`	recall last history line
`Ctl-C`		generate an interrupt signal (section 3.2.4, page 17)
`Ctl-D`		generate an end-of-file character (at the begin of the line)
`RETURN`		validate a line
`Esc-?`		display a summary of available commands

History: when a line is entered (i.e. terminated by `RETURN`), `linedit` records it in an internal list called history. It is later possible to recall history lines using appropriate commands (e.g. `Ctl-P` recall the last entered line) and to modify them as needed. It is also possible to recall a history line beginning with a given prefix. For instance to recall the previous line beginning with `write` simply type `write` followed by `Esc-P`. Another `Esc-P` will recall an earlier line beginning with `write`,...

Completion: another important feature of `linedit` is its completion facility. Indeed, `linedit` maintains a list of known words and uses it to complete the prefix of a word. Initially this list contains all predefined atoms and the atoms corresponding to available predicates. This list is dynamically updated when a new atom appears in the system (whether read at the top-level, created with a built-in predicate, associated to a new consulted predicate,...). When the completion key (`Tab`) is pressed `linedit` acts as follows:

- use the current word as a prefix.
- collect all words of the list that begin with this prefix.
- complete the current word with the longest common part of all matching words.
- if more than one word matches emit a beep (a second `Tab` will display all possibilities).

Example:

```
| ?- argu                (here the user presses Tab to complete the word)
| ?- argument_           (linedit completes argu with argument_ and emits a beep)
                         (the user presses again Tab to see all possible completions)
argument_counter         (linedit shows 3 possible completions)
argument_list
argument_value
| ?- argument_           (linedit redisplays the input line)

| ?- argument_c          (to select argument_counter the user presses c and Tab)
| ?- argument_counter    (linedit completes with argument_counter)
```

Finally, `linedit` allows the user to check that (square/curly) brackets are well balanced. For this, when a close bracket symbol, i.e. `)`, `]` or `}`, is typed, `linedit` determines the associated open bracket, i.e. `(`, `[` or `{`, and temporarily repositions the cursor on it to show the match.

3.3 Adjusting the size of Prolog stacks

GNU Prolog uses several stacks to execute a Prolog program. Each stack has a static size and cannot be dynamically increased during the execution. For each stack there is a default size but the user can define a new size by setting an environment variable. When a GNU Prolog program is run it first consults these variables and if they are not defined uses the default sizes. The following table presents each stack of GNU Prolog with its default size and the name of its associated environment variable:

Stack name	Default size (Kb)	Environment variable	Description
`local`	4096	`LOCALSZ`	control stack (environments and choice-points)
`global`	8192	`GLOBALSZ`	heap (compound terms)
`trail`	3072	`TRAILSZ`	conditional bindings (bindings to undo at backtracking)
`cstr`	3072	`CSTRSZ`	finite domain constraint stack (FD variables and constraints)

If the size of a stack is too small an overflow will occur during the execution. In that case GNU Prolog emits the following error message before stopping:

```
S stack overflow (size: N Kb, environment variable used: E)
```

where *S* is the name of the stack, *N* is the current stack size in Kb and *E* the name of the associated environment variable. When such a message occurs it is possible to (re)define the variable *E* with the new size. For instance to allocate 8192 Kb to the local stack under a Unix shell use:

```
LOCALSZ=8192; export LOCALS      (under sh or bash)
setenv LOCALSZ 8192              (under csh or tcsh)
```

This method allows the user to adjust the size of Prolog stacks. However, in some cases it is preferable not to allow the user to modify these sizes. For instance, when providing a stand alone executable whose behavior should be independent of the environment in which it is run. In that case the program should not consult environment variables and the programmer should be able to define new default stack sizes. The GNU Prolog compiler offers this facilities via several command-line options such as `--local-size` or `--fixed-sizes` (section 3.4.3, page 22).

Finally note that GNU Prolog stacks are virtually allocated (i.e. use virtual memory). This means that a physical memory page is allocated only when needed (i.e. when an attempt to read/write it occurs). Thus it is possible to define very large stacks. At the execution, only the needed amount of space will be physically allocated.

3.4 The GNU Prolog compiler

3.4.1 Different kinds of codes

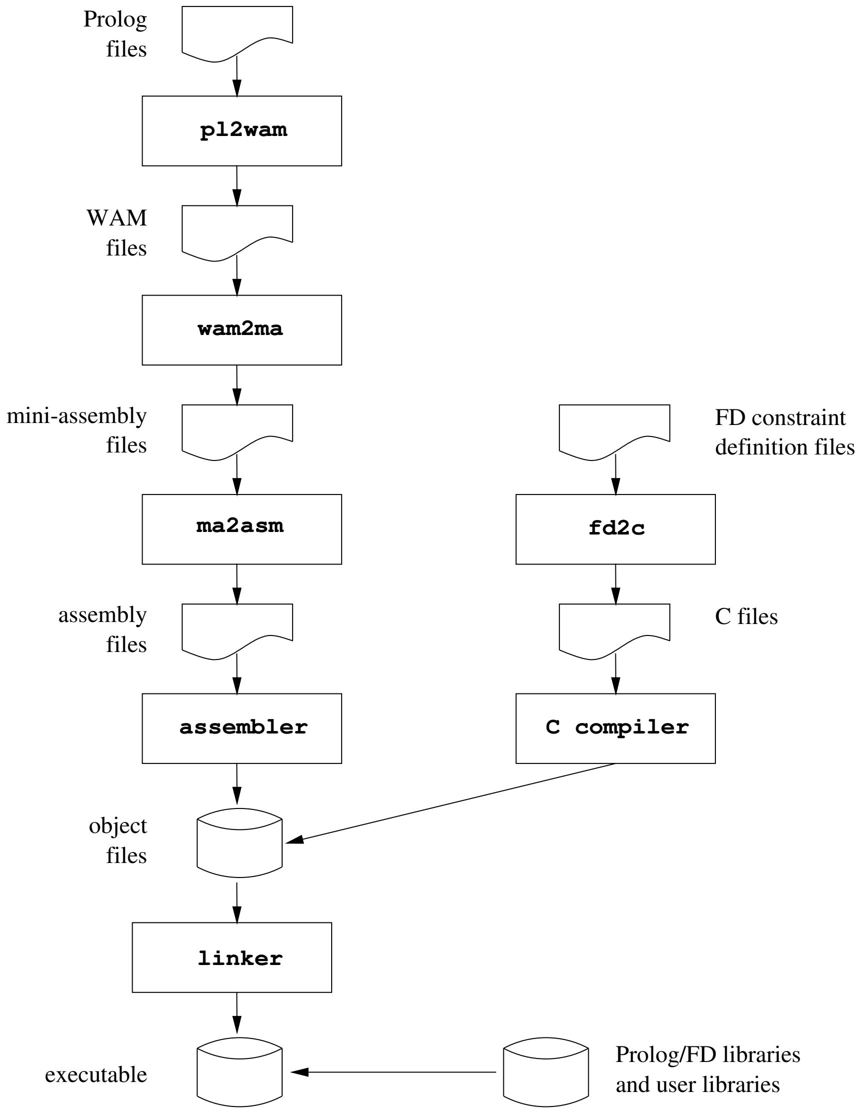
One of the main advantages of GNU Prolog is its ability to produce stand alone executables. A Prolog program can be compiled to native code to give rise to a machine-dependent executable using the GNU Prolog compiler. However native-code predicates cannot be listed nor fully debugged. So there is an alternative to native-code compilation: byte-code compilation. By default the GNU Prolog compiler produces native-code but via a command-line option it can produce a file ready for byte-code loading. This is exactly what `consult/1` does as was explained above (section 3.2.3, page 16). GNU Prolog also manages interpreted code using a Prolog interpreter written in Prolog. Obviously interpreted code is slower than byte-code but does not require the invocation of the GNU Prolog compiler. This interpreter is used each time a meta-call is needed as by `call/1` (section 6.2.3, page 47). This also the case of dynamically asserted clauses. The following table summarizes these three kinds of codes:

Type	Speed	Debug ?	For what
interpreted-code	slow	yes	meta-call and dynamically asserted clauses
byte-code	medium	yes	consulted predicates
native-code	fast	no	compiled predicates

3.4.2 Compilation scheme

Native-code compilation: a Prolog source is compiled in several stages to produce an object file that is linked to the GNU Prolog libraries to produce an executable. The Prolog source is first compiled to obtain a WAM [8] file. For a detailed study of the WAM the interested reader can refer to "Warren's Abstract Machine: A Tutorial Reconstruction"[18] [1]. The WAM file is translated to a machine-independent language specifically designed for GNU Prolog. This language is close to a (universal) assembly language and is based on a very reduced instruction set. For this reason this language is called mini-assembly (MA). The mini-assembly file is then mapped to the assembly language of the target machine. This assembly file is assembled to give rise to an object file which is then linked with the GNU Prolog libraries to provide an executable. The compiler also takes into account Finite Domain constraint definition files. It translates them to C and invoke the C compiler to obtain object files. The following figure presents this compilation scheme:

[18] `http://www.isg.sfu.ca/~hak/documents/wam.html`



Obviously all intermediate stages are hidden to the user who simply invokes the compiler on his Prolog file(s) (plus other files: C,...) and obtains an executable. However, it is also possible to stop the compiler at any given stage. This can be useful, for instance, to see the WAM code produced (perhaps when learning the WAM). Finally it is possible to give any kind of file to the compiler which will insert it in the compilation chain at the stage corresponding to its type. The type of a file is determined using the suffix of its file name. The following table presents all recognized types/suffixes:

Suffix of the file	Type of the file	Handled by:
`.pl`, `.pro`	Prolog source file	`pl2wam`
`.wam`	WAM source file	`wam2ma`
`.ma`	Mini-assembly source file	`ma2asm`
`.s`	Assembly source file	the assembler
`.c`, `.C`, `.CC`, `.cc`, `.cxx`, `.c++`, `.cpp`	C or C++ source file	the C compiler
`.fd`	Finite Domain constraint source file	`fd2c`
any other suffix (`.o`, `.a`,...)	any other type (object, library,...)	the linker (C linker)

Byte-code compilation: the same compiler can be used to compile a source Prolog file for byte-code. In that case the Prolog to WAM compiler is invoked using a specific option and produces a WAM for byte-code source file (suffixed `.wbc`) that can be later loaded using `load/1` (section 7.23.2, page 137). Note that this is exactly what `consult/1` (section 7.23.1, page 136) does as explained above (section 3.2.3, page 16).

3.4.3 Using the compiler

The GNU Prolog compiler is a command-line compiler similar in spirit to a Unix C compiler like `gcc`. To invoke the compiler use the `gplc` command as follows:

`% gplc [OPTION]... FILE...` (the `%` symbol is the operating system shell prompt)

The arguments of `gplc` are file names that are dispatched in the compilation scheme depending on the type determined from their suffix as was explained previously (section 3.4.2, page 20). All object files are then linked to produce an executable. Note however that GNU Prolog has no module facility (since there is not yet an ISO reference for Prolog modules) thus a predicate defined in a Prolog file is visible from any other predicate defined in any other file. GNU Prolog allows the user to split a big Prolog source into several files but does not offer any way to hide a predicate from others.

The simplest way to obtain an executable from a Prolog source file `prog.pl` is to use:

```
% gplc prog.pl
```

This will produce an native executable called `prog` which can be executed as follows:

```
% prog
```

However, there are several options that can be used to control the compilation:

General options:

`-o` *FILE*, `--output` *FILE*	use *FILE* as the name of the output file
`-W`, `--wam-for-native`	stop after producing WAM files(s)
`-w`, `--wam-for-byte-code`	stop after producing WAM for byte-code file(s) (force `--no-call-c`)
`-M`, `--mini-assembly`	stop after producing mini-assembly files(s)
`-S`, `--assembly`	stop after producing assembly files (s)
`-F`, `--fd-to-c`	stop after producing C files(s) from FD constraint definition file(s)
`-c`, `--object`	stop after producing object files(s)
`--temp-dir` *PATH*	use *PATH* as directory for temporary files
`--no-del-temp`	do not delete temporary files
`--no-decode-hexa`	do not decode hexadecimal predicate names
`-v`, `--verbose`	print executed commands
`-h`, `--help`	print a help and exit
`--version`	print version number and exit

Prolog to WAM compiler options:

`--pl-state` *FILE*	read *FILE* to set the initial Prolog state
`--no-susp-warn`	do not show warnings for suspicious predicates
`--no-singl-warn`	do not show warnings for named singleton variables
`--no-redef-error`	no not show errors for built-in predicate redefinitions
`--foreign-only`	only compile `foreign/1-2` directives
`--no-call-c`	do not allow the use of `fd_tell`, `'$call_c'`,...
`--no-inline`	do not inline predicates
`--no-reorder`	do not reorder predicate arguments
`--no-reg-opt`	do not optimize registers
`--min-reg-opt`	minimally optimize registers
`--no-opt-last-subterm`	do not optimize last subterm compilation
`--fast-math`	use fast mathematical mode (assume integer arithmetics)
`--keep-void-inst`	keep void WAM instructions in the output file
`--compile-msg`	print a compile message
`--statistics`	print statistics information

WAM to mini-assembly translator options:

`--comment`	include comments in the output file

Mini-assembly to assembly translator options:

`--comment`	include comments in the output file

C compiler options:

`--c-compiler` *`FILE`*	use *`FILE`* as C compiler
`-C` *`OPTION`*	pass *`OPTION`* to the C compiler

Assembler options:

`-A` *`OPTION`*	pass *`OPTION`* to the assembler

Linker options:

`--local-size` *`N`*	set default local stack size to *`N`* Kb
`--global-size` *`N`*	set default global stack size to *`N`* Kb
`--trail-size` *`N`*	set default trail stack size to *`N`* Kb
`--cstr-size` *`N`*	set default constraint stack size to *`N`* Kb
`--fixed-sizes`	do not consult environment variables at run-time (use default sizes)
`--no-top-level`	do not link the top-level (force `--no-debugger`)
`--no-debugger`	do not link the Prolog/WAM debugger
`--min-pl-bips`	link only used Prolog built-in predicates
`--min-fd-bips`	link only used FD solver built-in predicates
`--min-bips`	shorthand for: `--no-top-level --min-pl-bips --min-fd-bips`
`--min-size`	shorthand for: `--min-bips --strip`
`--no-fd-lib`	do not look for the FD library (maintenance only)
`-s`, `--strip`	strip the executable
`-L` *`OPTION`*	Pass *`OPTION`* to the linker

It is possible to only give the prefix of an option if there is no ambiguity.

The name of the output file is controlled via the `-o` *`FILE`* option. If present the output file produced will be named *`FILE`*. If not specified, the output file name depends on the last stage reached by the compiler. If the link is not done the output file name(s) is the input file name(s) with the suffix associated to the last stage. If the link is done, the name of the executable is the name (without suffix) of the first file name encountered in the command-line. Note that if the link is not done `-o` has no sense in the presence of multiple input file names. For this reason, several meta characters are available for substitution in *`FILE`*:

- `%f` is substitued by the whole input file name.
- `%F` is similar to `%f` but the directory part is omitted.
- `%p` is substitued by the whole prefix file name (omitting the suffix).
- `%P` is similar to `%p` but the directory part is omitted.
- `%s` is substitued by the file suffix (including the dot).
- `%d` is substitued by the directory part (empty if no directory is specified).
- `%c` is substitued by the value of an internal counter starting from 1 and auto-incremented.

By default the compiler runs in the native-code compilation scheme. To generate a WAM file for byte-code use the `--wam-for-byte-code` option. The resulting file can then be loaded using `load/1` (section 7.23.2, page 137).

To execute the Prolog to WAM compiler in a given *read environment* (operator definitions, character conversion table,...) use `--pl-state` *FILE*. The state file should be produced by `write_pl_state_file/1` (section 7.22.5, page 136).

By default the Prolog to WAM compiler inlines calls to some deterministic built-in predicates (e.g. `arg/3` and `functor/3`). Namely a call to such a predicate will not yield a classical predicate call but a simple C function call (which is obviously faster). It is possible to avoid this using `--no-inline`.

Another optimization performed by the Prolog to WAM compiler is unification reordering. The arguments of a predicate are reordered to optimize unification. This can be deactivated using `--no-reorder`. The compiler also optimizes the unification/loading of nested compound terms. More precisely, the compiler emits optimized instructions when the last subterm of a compound term is itself a compound term (e.g. lists). This can be deactivated using `--no-opt-last-subterm`.

By default the Prolog to WAM compiler fully optimizes the allocation of registers to decrease both the number of instruction produced and the number of used registers. A good allocation will generate many *void instructions* that are removed from the produced file except if `--keep-void-inst` is specified. To prevent any optimization use `--no-reg-opt` while `--min-reg-opt` forces the compiler to only perform simple register optimizations.

The Prolog to WAM compiler emits an error when a control construct or a built-in predicate is redefined. This can be avoided using `--no-redef-error`. The compiler also emits warnings for suspicious predicate definitions like `-/2` since this often corresponds to an earlier syntax error (e.g. `-` instead of _. This can be deactivated by specifying `--no-susp-warn`. Finally, the compiler warns when a singleton variable has a name (i.e. not the generic anonymous name _). This can be deactivated specifying `--no-singl-warn`.

Predicate names are encoded with an hexadecimal representation. This is explained in more detail later (section 3.4.6, page 26). By default the error messages from the linker (e.g. multiple definitions for a given predicate, reference to an undefined predicate,...) are filtered to replace any hexadecimal representation by the real predicate name. Specifying the `--no-decode-hexa` prevents `gplc` from filtering linker output messages and hexadecimal representations are then shown.

When producing an executable it is possible to specify default stack sizes (using `--`*STACK_NAME*`-size`) and to prevent it from consulting environment variables (using `--fixed-sizes`) as was explained above (section 3.3, page 19). By default the produced executable will include the top-level, the Prolog/WAM debugger and all Prolog and FD built-in predicates. It is possible to avoid linking the top-level (section 3.2, page 13) by specifying `--no-top-level`. In this case, at least one `initialization/1` directive (section 6.1.13, page 45) should be defined. The option `--no-debugger` does not link the debugger. To include only used built-in predicates that are actually used the options `--no-pl-bips` and/or `--no-fd-bips` can be specified. For the smallest executable all these options should be specified. This can be abbreviated by using the shorthand option `--min-bips`. By default, executables are not *stripped*, i.e. their symbol table is not removed. This table is only useful for the C debugger (e.g. when interfacing Prolog and C). To remove the symbol table (and then to reduce the size of the final executable) use `--strip`. Finally `--min-size` is a shortcut for `--min-bips` and `--strip`, i.e. the produced executable is as small as possible.

Example: compile and link two Prolog sources `prog1.pl` and `prog2.pl`. The resulting executable will be named `prog1` (since `-o` is not specified):

```
% gplc prog1.pl prog2.pl
```

Example: compile the Prolog file `prog.pl` to study basic WAM code. The resulting file will be named `prog.wam`:

```
% gplc -W --no-inline --no-reorder --keep-void-inst prog.pl
```

Example: compile the Prolog file `prog.pl` and its C interface file `utils.c` to provide an autonomous executable called `mycommand`. The executable is not stripped to allow the use of the C debugger:

```
% gplc -o mycommand prog.pl utils.c
```

Example: detail all steps to compile the Prolog file `prog.pl` (the resulting executable is stripped). All intermediate files are produced (`prog.wam`, `prog.ma`, `prog.s`, `prog.o` and the executable `prog`):

```
% gplc -W prog.pl
% gplc -M --comment prog.wam
% gplc -S --comment prog.ma
% gplc -c prog.s
% gplc -o prog -s prog.o
```

3.4.4 Running an executable

In this section we explain what happens when running an executable produced by the GNU Prolog native-code compiler. The default main function first starts the Prolog engine. This function collects all linked objects (issued from the compilation of Prolog files) and initializes them. The initialization of a Prolog object file consists in adding to appropriate tables new atoms, new predicates and executing its system directives. A system directive is generated by the Prolog to WAM compiler to reflect a (user) directive executed at compile-time such as `op/3` (section 6.1.10, page 44). Indeed, when the compiler encounters such a directive it immediately executes it and also generates a system directive to execute it at the start of the executable. When all system directives have been executed the Prolog engine executes all initialization directives defined with `initialization/1` (section 6.1.13, page 45). If several initialization directives appear in the same file they are executed in the order of appearance. If several initialization directives appear in different files the order in which they are executed is machine-dependant. However, on most machines the order will be the reverse order in which the associated files have been linked (this is not true under native win32). When all initialization directives have been executed the default main function looks for the GNU Prolog top-level. If present (i.e. it has been linked) it is called otherwise the program simply ends. Note that if the top-level is not linked and if there is no initialization directive the program is useless since it simply ends without doing any work. The default main function detects such a behavior and emits a warning message.

Example: compile an empty file `prog.pl` without linking the top-level and execute it:

```
% gplc --no-top-level prog.pl
% prog
Warning: no initial goal executed
   use a directive :- initialization(Goal)
   or remove the link option --no-top-level (or --min-bips or --min-size)
```

3.4.5 Generating a new interactive interpreter

In this section we show how to define a new top-level extending the GNU Prolog interactive interpreter with new predicate definitions. The obtained top-level can then be considered as an enriched version of the basic GNU Prolog top-level (section 3.2, page 13). Indeed, each added predicate can be viewed as a predefined predicate just like any other built-in predicate. This can be achieved by compiling these predicates and including the top-level at link-time.

The real question is: why would we include some predicates in a new top-level instead of simply consulting them under the GNU Prolog top-level ? There are two reasons for this:

- the predicate cannot be consulted. This is the case of a predicate calling foreign code, like a predicate interfacing with C (section 9, page 183) or a predicate defining a new FD constraint.
- the performance of the predicate is crucial. Since it is compiled to native-code such a predicate will be executed very quickly. Consulting will load it as byte-code. The gain is much more noticeable if

the program is run under the debugger. The included version will not be affected by the debugger while the consulted version will be several times slower. Obviously, a predicate should be included in a new top-level only when it is itself debugged since it is difficult to debug native-code.

To define a new top-level simply compile the set of desired predicates and linking them with the GNU Prolog top-level (this is the default) using `gplc` (section 3.4.3, page 22).

Example: let us define a new top-level called `my_top_level` including all predicates defined in `prog.pl`:

```
% gplc -o my_top_level prog.pl
```

By the way, note that if `prog.pl` is an empty Prolog file the previous command will simply create a new interactive interpreter similar to the GNU Prolog top-level.

Example: as before where some predicates of `prog.pl` call C functions defined in `utils.c`:

```
% gplc -o my_top_level prog.pl utils.c
```

In conclusion, defining a particular top-level is nothing else but a particular case of the native-code compilation. It is simple to do and very useful in practice.

3.4.6 The hexadecimal predicate name encoding

When the GNU Prolog compiler compiles a Prolog source to an object file it has to associate a symbol to each predicate name. However, the syntax of symbols is restricted to identifiers: string containing only letters, digits or underscore characters. On the other hand, predicate names (i.e. atoms) can contain any character with quotes if necessary (e.g. `'x+y=z'` is a valid predicate name). The compiler has then to encode predicate names respecting the syntax of identifiers. To achieve this, GNU Prolog uses an hexadecimal representation where each predicate name is translated to a symbol beginning with an `X` followed by the hexadecimal notation of the code of each character of the name.

Example: `'x+y=z'` will be encoded as `X782B793D7A` since `78` is the hexadecimal representation of the code of `x`, `2B` of the code of `+`, etc.

Since Prolog allows the user to define several predicates with the same name but with a different arity GNU Prolog encodes predicate indicators (predicate name followed by the arity). The symbol associated to the predicate name is then followed by an underscore and by the decimal notation of the arity.

Example: `'x+y=z'/3` will be encoded as `X782B793D7A_3`.

So, from the mini-assembly stage, each predicate indicator is replaced by its hexadecimal encoding. The knowledge of this encoding is normally not of interest for the user, i.e. the Prolog programmer. For this reason the GNU Prolog compiler hides this encoding. When an error occurs on a predicate (undefined predicate, predicate with multiple definitions,...) the compiler has to decode the symbol associated to the predicate indicator. For this `gplc` filters each message emitted by the linker to locate and decode eventual predicate indicators. This filtering can be deactivated specifying `--no-decode-hexa` when invoking `gplc` (section 3.4.3, page 22).

This filter is provided as an utility that can be invoked using the `hexgplc` command as follows:

`% hexgplc [`*`OPTION`*`]... `*`FILE`*`...` (the `%` symbol is the operating system shell prompt)

Options:

`--encode`	encoding mode (default mode is decoding)
`--relax`	decode also predicate names (not only predicate indicators)
`--printf` *FORMAT*	pass encoded/decoded string to C `printf(3)` with *FORMAT*
`--aux-father`	decode an auxiliary predicate as its father
`--aux-father2`	decode an auxiliary predicate as its father + auxiliary number
`--cmd-line`	encode/decode each argument of the command-line
`-H`	same as: `--cmd-line --encode`
`-P`	same as: `--cmd-line --relax`
`--help`	print a help and exit
`--version`	print version number and exit

It is possible to give a prefix of an option if there is no ambiguity.

Without arguments `hexgplc` runs in decoding mode reading its standard input and decoding each symbol corresponding to a predicate indicator. To use `hexgplc` in the encoding mode the `--encode` option must be specified. By default `hexgplc` only decodes predicate indicators, this can be relaxed using `--relax` to also take into account simple predicate names (the arity can be omitted). It is possible to format the output of an encoded/decoded string using `--printf` *FORMAT* in that case each string *S* is passed to the C `printf(3)` function as `printf(`*FORMAT*`,`*S*`)`.

Auxiliary predicates are generated by the Prolog to WAM compiler when simplifying some control constructs like `';'/2` present in the body of a clause. They are of the form `'$`*NAME*`/`*ARITY*`_$aux`*N*`'` where *NAME*`/`*ARITY* is the predicate indicator of the simplified (i.e. father) predicate and *N* is a sequential number (a predicate can give rise to several auxiliary predicates). It is possible to force `hexgplc` to decode an auxiliary predicate as its father predicate indicator using `--aux-father` or as its father predicate indicator followed by the sequential number using `--aux-father2`.

If no file is specified, `hexgplc` processes its standard input otherwise each file is treated sequentially. Specifying the `--cmd-line` option informs `hexgplc` that each argument is not a file name but a string that must be encoded (or decoded). This is useful to encode/decode a particular string. For this reason the option `-H` (encode to hexadecimal) and `-P` (decode to Prolog) are provided as shorthand. Then, to obtain the hexadecimal representation of a predicate *P* use:

```
% hexgplc -H P
```

Example:

```
% hexgplc -H 'x+y=z'
X782B793D7A
```

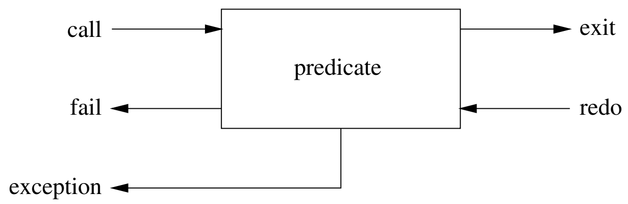
4 Debugging

4.1 Introduction

The GNU Prolog debugger provides information concerning the control flow of the program. The debugger can be fully used on consulted predicates (i.e. byte-code). For native compiled code only the calls/exits are traced, no internal behavior is shown. Under the debugger it is possible to exhaustively trace the execution or to set spy-points to only debug a specific part of the program. Spy-points allow the user to indicate on which predicates the debugger has to stop to allow the user to interact with it. The debugger uses the “procedure box control flow model”, also called the Byrd Box model since it is due to Lawrence Byrd.

4.2 The procedure box model

The procedure box model of Prolog execution provides a simple way to show the control flow. This model is very popular and has been adopted in many Prolog systems (e.g. SICStus Prolog, Quintus Prolog,...). A good introduction is the chapter 8 of “Programming in Prolog” of Clocksin & Mellish [2]. The debugger executes a program step by step tracing an invocation to a predicate (`call`) and the return from this predicate due to either a success (`exit`) or a failure (`fail`). When a failure occurs the execution backtracks to the last predicate with an alternative clause. The predicate is then re-invoked (`redo`). Another source of change of the control flow is due to exceptions. When an exception is raised from a predicate (`exception`) by `throw/1` (section 6.2.4, page 47) the control is given back to the most recent predicate that has defined a handler to recover this exception using `catch/3` (section 6.2.4, page 47). The procedure box model shows these different changes in the control flow, as illustrated here:



Each arrow corresponds to a *port*. An arrow to the box indicates that the control is given to this predicate while an arrow from the box indicates that the control is given back from the procedure. This model visualizes the control flow through these five ports and the connections between the boxes associated to subgoals. Finally, it should be clear that a box is associated to one invocation of a given predicate. In particular, a recursive predicate will give raise to a box for each invocation of the predicate with different entries/exits in the control flow. Since this might get confusing for the user, the debugger associates to each box a unique identifier (i.e. the invocation number).

4.3 Debugging predicates

4.3.1 Running and stopping the debugger

`trace/0` activates the debugger. The next invocation of a predicate will be traced.

`debug/0` activates the debugger. The next invocation of a predicate on which a spy-point has been set will be traced.

It is important to understand that the information associated to the control flow is only available when the debugger is on. For efficiency reasons, when the debugger is off the information concerning the control flow (i.e. the boxes) is not retained. So, if the debugger is activated in the middle of a computation (by a call to `debug/0` or `trace/0` in the program or after the interrupt key sequence (`Ctl-C`) by choosing `trace` or `debug`), information prior to this point is not available.

`debugging/0`: prints onto the terminal information about the current debugging state (whether the debugger is switched on, what are the leashed ports, spy-points defined,...).

`notrace/0` or `nodebug/0` switches the debugger off.

`wam_debug/0` invokes the sub-debugger devoted to the WAM data structures (section 4.6, page 32). It can be also invoked using the `W` debugger command (section 4.5, page 31).

4.3.2 Leashing ports

`leash(Ports)` requests the debugger to prompt the user, as he creeps through the program, for every port defined in the `Ports` list. Each element of `Ports` is an atom in `call`, `exit`, `redo`, `fail`, `exception`. `Ports` can also be an atom defining a shorthand:

- `full`: equivalent to `[call, exit, redo, fail, exception]`
- `half`: equivalent to `[call, redo]`
- `loose`: equivalent to `[call]`
- `none`: equivalent to `[]`
- `tight`: equivalent to `[call, redo, fail, exception]`

When an unleashed port is encountered the debugger continues to show the associated goal but does not stop the execution to prompt the user.

4.3.3 Spy-points

When dealing with big sources it is not very practical to creep through the entire program. It is preferable to define a set of spy-points on interesting predicates to be prompted when the debugger reaches one of these predicates. Spy-points can be added either using `spy/1` (or `spypoint_condition/3`) or dynamically when prompted by the debugger using the `+` (or `*`) debugger command (section 4.5, page 31). The current mode of leashing does not affect spy-points in the sense that user interaction is requested on every port.

`spy(PredSpec)` sets a spy-point on all the predicates given by `PredSpec`. `PredSpec` defines one or several predicates and has one of the following forms:

- `[PredSpec1, PredSpec2,...]`: set a spy-point for each element of the list.
- `Name`: set a spy-point for any predicate whose name is `Name` (whatever the arity).
- `Name/Arity`: set a spy-point for the predicate whose name is `Name` and arity is `Arity`.
- `Name/A1-A2`: set a spy-point for the each predicate whose name is `Name` and arity is between `A1` and `A2`.

It is not possible to set a spy-point on an undefined predicate.

The following predicate is used to remove one or several spy-points:

`nospy(PredSpec)` removes the spy-points from the specified predicates.

`nospyall/0` removes all spy-points:

It is also possible to define conditional spy-points.

`spypoint_condition(Goal, Port, Test)` sets a conditional spy-point on the predicate for `Goal`. When the debugger reaches a conditional spy-point it only shows the associated goal if the following conditions are verified:

- the actual goal unifies with `Goal`.
- the actual port unifies with `Port`.
- the Prolog goal `Test` succeeds.

4.4 Debugging messages

We here described which information is displayed by the debugger when it shows a goal. The basic format is as follows:

`S N M Port: Goal ?`

`S` is a spy-point indicator: if there is a spy-point on the current goal the `+` symbol is displayed else a space is displayed. *`N`* is the invocation number. This unique number can be used to correlate the trace messages for the various ports, since it is unique for every invocation. *`M`* is an index number which represents the number of direct ancestors of the goal (i.e. the current depth of the goal). *`Port`* specifies the particular port (`call`, `exit`, `fail`, `redo`, `exception`). *`Goal`* is the current goal (it is then possible to inspect its current instantiation) which is displayed using `write_term/3` with `quoted(true)` and `max_depth(D)` options (section 7.14.6, page 96). Initially *`D`* (the print depth) is set to 10 but can be redefined using the `<` debugger command (section 4.5, page 31). The `?` symbol is displayed when the debugger is waiting a command from the user. (i.e. `Port` is a leashed port). If the port is unleashed, this symbol is not displayed and the debugger continues the execution displaying the next goal.

4.5 Debugger commands

When the debugger reaches a leashed port it shows the current goal followed by the `?` symbol. At this point there are many commands available. Typing `RETURN` will creep into the program. Continuing to creep will show all the control flow. The debugger shows every port for every predicate encountered during the execution. It is possible to select the ports at which the debugger will prompt the user using the built-in predicate `leash/1` (section 4.3.2, page 30). Each command is only one character long:

Command	Name	Description
`RET` or `c`	creep	single-step to the next port
`l`	leap	continue the execution only stopping when a goal with a spy-point is reached
`s`	skip	skip over the entire execution of the current goal. No message will be shown until control returns
`G`	go to	ask for an invocation number and continue the execution until a port is reached for that invocation number
`r`	retry	try to restart the invocation of the current goal by failing until reaching the invocation of the goal. The state of execution is the same as when the goal was initially invoked (except when using side-effect predicates)
`f`	fail	force the current goal to fail immediately
`w`	write	show the current goal using `write/2` (section 7.14.6, page 96)
`d`	display	show the current goal using `display/2` (section 7.14.6, page 96)
`p`	print	show the current goal using `print/2` (section 7.14.6, page 96)
`e`	exception	show the pending exception. Only applicable to an `exception` port
`g`	ancestors	show the list of ancestors of the current goal
`A`	alternatives	show the list of ancestors of the current goal combined with choice-points
`u`	unify	ask for a term and unify the current goal with this term. This is convenient for getting a specific solution. Only available at a `call` port
`.`	father file	show the Prolog file name and the line number where the current predicate is defined
`n`	no debug	switch the debugger off. Same as `nodebug/0` (section 4.3.1, page 29)
`=`	debugging	show debugger information. Same as `debugging/0` (section 4.3.1, page 29)
`+`	spy this	set a spy-point on the current goal. Uses `spy/1` (section 4.3.3, page 30)
`-`	nospy this	remove a spy-point on the current goal. Uses `nospy/1` (section 4.3.3, page 30)
`*`	spy conditionally	ask for a term `Goal, Port, Test` (terminated by a dot) and set a conditional spy-point on the current predicate. `Goal` and the current goal must have the same predicate indicator. Uses `spypoint_condition/3` (section 4.3.3, page 30)
`L`	listing	list the clauses associated to the current predicate. Uses `listing/1` (section 7.23.3, page 138)
`a`	abort	abort the current execution. Same as `abort/0` (section 7.18.1, page 112)
`b`	break	invoke a recursive top-level. Same as `break/0` (section 7.18.1, page 112)
`@`	execute goal	ask for a goal and execute it
`<`	set print depth	ask for an integer and set the print depth to this value (`-1` for no depth limit)
`h` or `?`	help	display a summary of available commands
`W`	WAM debugger	invoke the low-level WAM debugger (section 4.6, page 32)

4.6 The WAM debugger

In some cases it is interesting to have access to the WAM data structures. This sub-debugger allows the user to inspect/modify the contents of any stack or register of the WAM. The WAM debugger is invoked using the built-in predicate `wam_debug/0` (section 4.3.1, page 29) or the `W` debugger command (section 4.5, page 31). The following table presents the specific commands of the WAM debugger:

Command	Description
`write` *A* [*N*]	write *N* terms starting at the address *A* using `write/1` (section 7.14.6, page 96)
`data` *A* [*N*]	display *N* words starting at the address *A*
`modify` *A* [*N*]	display and modify *N* words starting at the address *A*
`where` *A*	display the real address corresponding to *A*
`what` *RA*	display what corresponds to the real address *RA*
`deref` *A*	display the dereferenced word starting at the address *A*
`envir` [*SA*]	display the contents of the environment located at *SA* (or the current one)
`backtrack` [*SA*]	display the contents of the choice-point located at *SA* (or the current one)
`backtrack all`	display all choice-points
`quit`	quit the WAM debugger
`help`	display a summary of available commands

In the above table the following conventions apply:

- elements between [and] are optional.
- *N* is an optional integer (defaults to 1).
- *A* is a WAM address, its syntax is: *BANK_NAME* [`[` *N* `]`], i.e. a bank name possibly followed by an index (defaults to 0). *BANK_NAME* is either:
  - `reg`: WAM general register (stack pointers, continuation, ...).
  - `x`: WAM X register (temporary variables, i.e. arguments).
  - `y`: WAM Y register (permanent variables).
  - `ab`: WAM X register saved in the current choice-point.
  - *STACK_NAME*: WAM stack (*STACK_NAME* in `local`, `global`, `trail`, `cstr`).
- *SA* is a WAM stack address, i.e. *STACK_NAME* [`[` *N* `]`] (special case of WAM addresses).
- *RA* is a real address, its syntax is the syntax of C integers (in particular the notation `0x...` is recognized).

It is possible to only use the first letters of a commands and bank names when there is no ambiguity. Also the square brackets `[ ]` enclosing the index of a bank name can be omitted. For instance the following command (showing the contents of 25 consecutive words of the global stack from the index 3): `data global[3] 25` can be abbreviated as: `d g 3 25`.

5 Format of definitions

5.1 General format

The definition of control constructs, directives and built-in predicates is presented as follows:

Templates

Specifies the types of the arguments and which of them shall be instantiated (mode). Types and modes are described later (section 5.2, page 35).

Description

Describes the behavior (in the absence of any error conditions). It is explicitly mentioned when a built-in predicate is re-executable on backtracking. Predefined operators involved in the definition are also mentioned.

Errors

Details the error conditions. Possible errors are detailed later (section 5.3, page 37). For directives, this part is omitted.

Portability

Specifies whether the definition conforms to the ISO standard or is a GNU Prolog extension.

5.2 Types and modes

The templates part defines, for each argument of the concerned built-in predicate, its mode and type. The mode specifies whether or not the argument must be instantiated when the built-in predicate is called. The mode is encoded with a symbol just before the type. Possible modes are:

- `+`: the argument must be instantiated.
- `-`: the argument must be a variable (will be instantiated if the built-in predicate succeeds).
- `?`: the argument can be instantiated or a variable.

The type of an argument is defined by the following table:

Type	Description
*TYPE*_list	a list whose the type of each element is *TYPE*
*TYPE1*_or_*TYPE2*	a term whose type is either *TYPE1* or *TYPE2*
atom	an atom
atom_property	an atom property (section 7.19.12, page 120)
boolean	the atom true or false
byte	an integer ≥ 0 and ≤ 255
callable_term	an atom or a compound term
character	a single character atom
character_code	an integer ≥ 1 and ≤ 255
clause	a clause (fact or rule)
close_option	a close option (section 7.10.7, page 72)
compound_term	a compound term
evaluable	an arithmetic expression (section 7.6.1, page 57)
fd_bool_evaluable	a boolean FD expression (section 8.7.1, page 173)
fd_labeling_option	an FD labeling option (section 8.9.1, page 179)
fd_evaluable	an arithmetic FD expression (section 8.6.1, page 170)
fd_variable	an FD variable
flag	a Prolog flag (section 7.22.1, page 133)
float	a floating point number
head	a head of a clause (atom or compound term)
integer	an integer
in_byte	an integer ≥ 0 and ≤ 255 or -1 (for the end-of-file)
in_character	a single character atom or the atom end_of_file (for the end-of-file)
in_character_code	an integer ≥ 1 and ≤ 255 or -1 (for the end-of-file)
io_mode	an atom in: read, write or append
list	the empty list [] or a non-empty list [_|_]
nonvar	any term that is not a variable
number	an integer or a floating point number
operator_specifier	an operator specifier (section 7.14.10, page 100)
os_file_property	an operating system file property (section 7.27.11, page 147)
predicate_indicator	a term Name/Arity where Name is an atom and Arity an integer ≥ 0. A callable term can be given if the strict_iso Prolog flag is switched off (section 7.22.1, page 133)
predicate_property	a predicate property (section 7.8.2, page 65)
read_option	a read option (section 7.14.1, page 92)
socket_address	a term of the form 'AF_UNIX'(A) or 'AF_INET'(A,N) where A is an atom and N an integer
socket_domain	an atom in: 'AF_UNIX' or 'AF_INET'
source_sink	an atom identifying a source or a sink
stream	a stream-term: a term of the form '$stream'(N) where N is an integer ≥ 0
stream_option	a stream option (section 7.10.6, page 70)
stream_or_alias	a stream-term or an alias (atom)
stream_position	a stream position: a term '$stream_position'(I1, I2, I3, I4) where I1, I2, I3 and I4 are integers
stream_property	a stream property (section 7.10.10, page 73)
stream_seek_method	an atom in: bof, current or eof
term	any term
var_binding_option	a variable binding option (section 7.5.3, page 56)
write_option	a write option (section 7.14.6, page 96)

5.3 Errors

5.3.1 General format and error context

When an error occurs an exception of the form: `error(`*`ErrorTerm`*`, `*`Caller`*`)` is raised. *ErrorTerm* is a term specifying the error (detailed in next sections) and *Caller* is a term specifying the context of the error. The context is either the predicate indicator of the last invoked built-in predicate or an atom giving general context information.

Using exceptions allows the user both to recover an error using `catch/3` (section 6.2.4, page 47) and to raise an error using `throw/1` (section 6.2.4, page 47).

To illustrate how to write error cases, let us write a predicate `my_pred(X)` where `X` must be an integer:

```
my_pred(X) :-
        (   nonvar(X) ->
            true
        ;   throw(error(instantiation_error), my_pred/1)),
        ),
        (   integer(X) ->
            true
        ;   throw(error(type_error(integer, X), my_pred/1))
        ),
        ...
```

To help the user to write these error cases, a set of system predicates is provided to raise errors. These predicates are of the form `'$pl_err_...'` and they all refer to the implicit error context. The predicates `set_bip_name/2` (section 7.22.3, page 135) and `current_bip_name/2` (section 7.22.4, page 135) are provided to set and recover the name and the arity associated to this context (an arity < 0 means that only the atom corresponding to the functor is significant). Using these system predicates the user could define the above predicate as follow:

```
my_pred(X) :-
        set_bip_name(my_pred,1),
        (   nonvar(X) ->
            true
        ;   '$pl_err_instantiation'
        ),
        (   integer(X) ->
            true
        ;   '$pl_err_type'(integer, X)
        ),
        ...
```

The following sections detail each kind of errors (and associated system predicates).

5.3.2 Instantiation error

An instantiation error occurs when an argument or one of its components is variable while an instantiated argument was expected. *ErrorTerm* has the following form: `instantiation_error`.

The system predicate `'$pl_err_instantiation'` raises this error in the current error context (section 5.3.1, page 37).

5.3.3 Type error

A type error occurs when the type of an argument or one of its components is not the expected type (but not a variable). *ErrorTerm* has the following form: `type_error(`*`Type`*`, Culprit)` where *Type* is the expected type and *Culprit* the argument which caused the error. *Type* is one of:

- `atom`
- `atomic`
- `boolean`
- `byte`
- `callable`
- `character`
- `compound`
- `evaluable`
- `fd_bool_evaluable`
- `fd_evaluable`
- `fd_variable`
- `float`
- `in_byte`
- `in_character`
- `integer`
- `list`
- `number`
- `predicate_indicator`
- `variable`

The system predicate `'$pl_err_type'(Type, Culprit)` raises this error in the current error context (section 5.3.1, page 37).

5.3.4 Domain error

A domain error occurs when the type of an argument is correct but its value is outside the expected domain. *ErrorTerm* has the following form: `domain_error(`*`Domain`*`, `*`Culprit`*`)` where *Domain* is the expected domain and *Culprit* the argument which caused the error. *Domain* is one of:

- `atom_property`
- `buffering_mode`
- `character_code_list`
- `close_option`
- `date_time`
- `eof_action`
- `fd_labeling_option`
- `flag_value`
- `format_control_sequence`
- `g_array_index`
- `io_mode`
- `non_empty_list`
- `not_less_than_zero`
- `operator_priority`
- `operator_specifier`
- `os_file_permission`
- `os_file_property`
- `os_path`
- `predicate_property`
- `prolog_flag`
- `read_option`
- `selectable_item`
- `socket_address`
- `socket_domain`
- `source_sink`
- `statistics_key`
- `statistics_value`
- `stream`
- `stream_option`
- `stream_or_alias`
- `stream_position`
- `stream_property`
- `stream_seek_method`
- `stream_type`
- `term_stream_or_alias`
- `var_binding_option`
- `write_option`

The system predicate `'$pl_err_domain'(Domain, Culprit)` raises this error in the current error context (section 5.3.1, page 37).

5.3.5 Existence error

an existence error occurs when an object on which an operation is to be performed does not exist. *ErrorTerm* has the following form: `existence_error(`*Object*`, `*Culprit*`)` where *Object* is the type of the object and *Culprit* the argument which caused the error. *Object* is one of:

- `procedure`
- `source_sink`
- `stream`

The system predicate `'$pl_err_existence'(Object, Culprit)` raises this error in the current error context (section 5.3.1, page 37).

5.3.6 Permission error

A permission error occurs when an attempt to perform a prohibited operation is made. *ErrorTerm* has the following form: `permission_error(`*Operation*`, `*Permission*`, `*Culprit*`)` where *Operation* is the operation which caused the error, *Permission* the type of the tried permission and *Culprit* the argument which caused the error. *Operation* is one of:

- `access`
- `add_alias`
- `close`
- `create`
- `input`
- `modify`
- `open`
- `output`
- `reposition`

and *Permission* is one of:

- `binary_stream`
- `flag`
- `operator`
- `past_end_of_stream`
- `private_procedure`
- `source_sink`
- `static_procedure`
- `stream`
- `text_stream`

The system predicate `'$pl_err_permission'(Operation, Permission, Culprit)` raises this error in the current error context (section 5.3.1, page 37).

5.3.7 Representation error

A representation error occurs when an implementation limit has been breached. *ErrorTerm* has the following form: `representation_error(`*Limit*`)` where *Limit* is the name of the reached limit. *Limit* is one of:

- `character`
- `character_code`
- `in_character_code`
- `max_arity`
- `max_integer`
- `min_integer`
- `too_many_variables`

The errors `max_integer` and `min_integer` are not currently implemented.

The system predicate `'$pl_err_representation'(Limit)` raises this error in the current error context (section 5.3.1, page 37).

5.3.8 Evaluation error

An evaluation error occurs when an arithmetic expression gives rise to an exceptional value. *ErrorTerm* has the following form: `evaluation_error(`*Error*`)` where *Error* is the name of the error. *Error* is one of:

- `float_overflow`
- `int_overflow`
- `undefined`
- `underflow`
- `zero_divisor`

The errors `float_overflow`, `int_overflow`, `undefined` and `underflow` are not currently implemented.

The system predicate `'$pl_err_evaluation'(Error)` raises this error in the current error context (section 5.3.1, page 37).

5.3.9 Resource error

A resource error occurs when GNU Prolog does not have enough resources. *ErrorTerm* has the following form: `resource_error(`*Resource*`)` where *Resource* is the name of the resource. *Resource* is one of:

- `print_object_not_linked`
- `too_big_fd_constraint`
- `too_many_open_streams`

The system predicate `'$pl_err_resource'(Resource)` raises this error in the current error context (section 5.3.1, page 37).

5.3.10 Syntax error

A syntax error occurs when a sequence of character does not conform to the syntax of terms. *ErrorTerm* has the following form: `syntax_error(`*Error*`)` where *Error* is an atom explaining the error.

The system predicate `'$pl_err_syntax'(Error)` raises this error in the current error context (section 5.3.1, page 37).

5.3.11 System error

A system error can occur at any stage. A system error is generally associated to an external component (e.g. operating system). *ErrorTerm* has the following form: `system_error(`*Error*`)` where *Error* is an atom explaining the error. This is an extension to ISO which only defines `system_error` without arguments.

The system predicate `'$pl_err_system'(Error)` raises this error in the current error context (section 5.3.1, page 37).

6 Prolog directives and control constructs

6.1 Prolog directives

6.1.1 Introduction

Prolog directives are annotations inserted in Prolog source files for the compiler. A Prolog directive is used to specify:

- the properties of some procedures defined in the source file.
- the format and the syntax for read-terms in the source file (using changeable Prolog flags).
- included source files.
- a goal to be executed at run-time.

6.1.2 `dynamic/1`

Templates

```
dynamic(+predicate_indicator)
dynamic(+predicate_indicator_list)
dynamic(+predicate_indicator_sequence)
```

Description

`dynamic(Pred)` specifies that the procedure whose predicate indicator is `Pred` is a dynamic procedure. This directive makes it possible to alter the definition of `Pred` by adding or removing clauses. For more information refer to the section about dynamic clause management (section 7.7.1, page 60).

This directive shall precede the definition of `Pred` in the source file.

If there is no clause for `Pred` in the source file, `Pred` exists however as an empty predicate (this means that `current_predicate(Pred)` succeeds).

In order to allow multiple definitions, `Pred` can also be a list of predicate indicators or a sequence of predicate indicators using `','/2` as separator.

Portability

ISO directive.

6.1.3 `public/1`

Templates

```
public(+predicate_indicator)
public(+predicate_indicator_list)
public(+predicate_indicator_sequence)
```

Description

`public(Pred)` specifies that the procedure whose predicate indicator is `Pred` is a public procedure. This directive makes it possible to inspect the clauses of `Pred`. For more information refer to the section about dynamic clause management (section 7.7.1, page 60).

This directive shall precede the definition of `Pred` in the source file. Since a dynamic procedure is also public. It is useless (but correct) to define a public directive for a predicate already declared as dynamic.

In order to allow multiple definitions, `Pred` can also be a list of predicate indicators or a sequence of predicate indicators using `','/2` as separator.

Portability

GNU Prolog directive. The ISO reference does not define any directive to declare a predicate public but it does distinguish public predicates. It is worth noting that in most Prolog systems the `public/1` directive is as a visibility declaration. Indeed, declaring a predicate as public makes it visible from any predicate defined in any other file (otherwise the predicate is only visible from predicates defined in the same source file as itself). When a module system is incorporated in GNU Prolog a more general visibility declaration shall be provided conforming to the ISO reference.

6.1.4 `multifile/1`

Templates

```
multifile(+predicate_indicator)
multifile(+predicate_indicator_list)
multifile(+predicate_indicator_sequence)
```

Description

`multifile(Pred)` is not supported by GNU Prolog. When such a directive is encountered it is simply ignored. All clauses for a given predicate must reside in a single file.

Portability

ISO directive. Not supported.

6.1.5 `discontiguous/1`

Templates

```
discontiguous(+predicate_indicator)
discontiguous(+predicate_indicator_list)
discontiguous(+predicate_indicator_sequence)
```

Description

`discontiguous(Pred)` specifies that the procedure whose predicate indicator is `Pred` is a discontiguous procedure. Namely, the clauses defining `Pred` are not restricted to be consecutive but can appear anywhere in the source file.

This directive shall precede the definition of `Pred` in the source file.

In order to allow multiple definitions, `Pred` can also be a list of predicate indicators or a sequence of predicate indicators using `','/2` as separator.

Portability

ISO directive. The ISO reference document states that if there is no clause for `Pred` in the source file, `Pred` exists however as an empty predicate (i.e. `current_predicate(Pred)` will succeed). This is not the case for GNU Prolog.

6.1.6 ensure_linked/1

Templates

```
ensure_linked(+predicate_indicator)
ensure_linked(+predicate_indicator_list)
ensure_linked(+predicate_indicator_sequence)
```

Description

`ensure_linked(Pred)` specifies that the procedure whose predicate indicator is `Pred` must be included by the linker. This directive is useful when compiling to native code to force the linker to include the code of a given predicate. Indeed, if the `gplc` is invoked with an option to reduce the size of the executable (section 3.4.3, page 22), the linker only includes the code of predicates that are statically referenced. However, the linker cannot detect dynamically referenced predicates (used as data passed to a meta-call predicate). The use of this directive prevents it to exclude the code of such predicates.

In order to allow multiple definitions, `Pred` can also be a list of predicate indicators or a sequence of predicate indicators using `','/2` as separator.

Portability

GNU Prolog directive.

6.1.7 built_in/0, built_in/1, built_in_fd/0, built_in_fd/1

Templates

```
built_in
built_in(+predicate_indicator)
built_in(+predicate_indicator_list)
built_in(+predicate_indicator_sequence)
built_in_fd
built_in_fd(+predicate_indicator)
built_in_fd(+predicate_indicator_list)
built_in_fd(+predicate_indicator_sequence)
```

Description

`built_in` specifies that the procedures defined from now have the `built_in` property (section 7.8.2, page 65).

`built_in(Pred)` is similar to `built_in/0` but only affects the procedure whose predicate indicator is `Pred`.

This directive shall precede the definition of `Pred` in the source file.

In order to allow multiple definitions, `Pred` can also be a list of predicate indicators or a sequence of predicate indicators using `','/2` as separator.

`built_in_fd` (resp. `built_in_fd(Pred)`) is similar to `built_in` (resp. `built_in(Pred)`) but sets the `built_in_fd` predicate property (section 7.8.2, page 65).

Portability

GNU Prolog directives.

6.1.8 `include/1`

Templates

`include(+atom)`

Description

`include(File)` specifies that the content of the Prolog source `File` shall be inserted. The resulting Prolog text is identical to the Prolog text obtained by replacing the directive by the content of the Prolog source `File`.

See `absolute_file_name/2` for information about the syntax of `File` (section 7.26.1, page 141).

Portability

ISO directive.

6.1.9 `ensure_loaded/1`

Templates

`ensure_loaded(+atom)`

Description

`ensure_loaded(File)` is not supported by GNU Prolog. When such a directive is encountered it is simply ignored.

Portability

ISO directive. Not supported.

6.1.10 `op/3`

Templates

`op(+integer, +operator_specifier, +atom_or_atom_list)`

Description

`op(Priority, OpSpecifier, Operator)` alters the operator table. This directive is executed as soon as it is encountered by calling the built-in predicate `op/3` (section 7.14.10, page 100). A system directive is also generated to reflect the effect of this directive at run-time (section 3.4.4, page 25).

Portability

ISO directive.

6.1.11 `char_conversion/2`

Templates

`char_conversion(+character, +character)`

Description

`char_conversion(InChar, OutChar)` alters the character-conversion mapping. This directive is executed as soon as it is encountered by a call to the built-in predicate `char_conversion/2` (section 7.14.12, page 102). A system directive is also generated to reflect the effect of this directive at run-time (section 3.4.4, page 25).

Portability

ISO directive.

6.1.12 `set_prolog_flag/2`

Templates

```
set_prolog_flag(+flag, +term)
```

Description

`set_prolog_flag(Flag, Value)` sets the value of the Prolog flag `Flag` to `Value`. This directive is executed as soon as it is encountered by a call to the built-in predicate `set_prolog_flag/2` (section 7.22.1, page 133). A system directive is also generated to reflect the effect of this directive at run-time (section 3.4.4, page 25).

Portability

ISO directive.

6.1.13 `initialization/1`

Templates

```
initialization(+callable_term)
```

Description

`initialization(Goal)` adds `Goal` to the set of goal which shall be executed at run-time. A user directive is generated to execute `Goal` at run-time. If several initialization directives appear in the same file they are executed in the order of appearance (section 3.4.4, page 25).

Portability

ISO directive.

6.1.14 `foreign/2`, `foreign/1`

Templates

```
foreign(+callable_term, +foreign_option_list)
foreign(+callable_term)
```

Description

`foreign(Template, Options)` defines an interface predicate whose prototype is `Template` according to the options given by `Options`. Refer to the foreign code interface for more information (section 9.1, page 183).

`foreign(Template)` is equivalent to `foreign(Template, [])`.

Portability

GNU Prolog directive.

6.2 Prolog control constructs

6.2.1 `true/0`, `fail/0`, `!/0`

Templates

```
true
fail
!
```

Description

`true` always succeeds.

`fail` always fails (enforces backtracking).

`!` always succeeds and the for side-effect of removing all choice-points created since the invocation of the predicate activating it.

Errors

None.

Portability

ISO control constructs.

6.2.2 `(',')/2` - conjunction, `(;)/2` - disjunction, `(->)/2` - if-then

Templates

```
','(+callable_term, +callable_term)
;(+callable_term, +callable_term)
->(+callable_term, +callable_term)
```

Description

`Goal1 , Goal2` executes `Goal1` and, in case of success, executes `Goal2`.

`Goal1 ; Goal2` first creates a choice-point and executes `Goal1`. On backtracking `Goal2` is executed.

`Goal1 -> Goal2` first executes `Goal1` and, in case of success, removes all choice-points created by `Goal1` and executes `Goal2`. This control construct acts like an if-then (`Goal1` is the test part and `Goal2` the then part). Note that if `Goal1` fails `->/2` fails also. `->/2` is often combined with `;/2` to define an if-then-else as follows: `Goal1 -> Goal2 ; Goal3`. Note that `Goal1 -> Goal2` is the first argument of the `(;)/2` and `Goal3` (the else part) is the second argument. Such an if-then-else control construct first creates a choice-point for the else-part (intuitively associated to `;/2`) and then executes `Goal1`. In case of success, all choice-points created by `Goal1` together with the choice-point for the else-part are removed and `Goal2` is executed. If `Goal1` fails then `Goal3` is executed.

`','`, `;` and `->` are predefined infix operators (section 7.14.10, page 100).

Errors

`Goal1` or `Goal2` is a variable	`instantiation_error`
`Goal1` is neither a variable nor a callable term	`type_error(callable, Goal1)`
`Goal2` is neither a variable nor a callable term	`type_error(callable, Goal2)`
The predicate indicator `Pred` of `Goal1` or `Goal2` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`

Portability

ISO control constructs.

6.2.3 `call/1`

Templates

```
call(+callable_term)
```

Description

`call(Goal)` executes `Goal`. `call/1` succeeds if `Goal` represents a goal which is true. When `Goal` contains a cut symbol `!` (section 6.2.1, page 46) as a subgoal, the effect of `!` does not extend outside `Goal`.

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`

Portability

ISO control construct.

6.2.4 `catch/3`, `throw/1`

Templates

```
catch(?callable_term, ?term, ?term)
throw(+nonvar)
```

Description

`catch(Goal, Catcher, Recovery)` is similar to `call(Goal)` (section 6.2.3, page 47). If this succeeds or fails, so does the call to `catch/3`. If however, during the execution of `Goal`, there is a call to `throw(Ball)`, the current flow of control is interrupted, and control returns to a call of `catch/3` that is being executed. This can happen in one of two ways:

- implicitly, when an error condition for a built-in predicate is satisfied.
- explicitly, when the program executes a call of `throw/1` because the program wishes to abandon the current processing, and instead to take an alternative action.

`throw(Ball)` causes the normal flow of control to be transferred back to an existing call of `catch/3`. When a call to `throw(Ball)` happens, `Ball` is copied and the stack is unwound back to the call to `catch/3`, whereupon the copy of `Ball` is unified with `Catcher`. If this unification succeeds, then `catch/3` executes the goal `Recovery` using `call/1` (section 6.2.3, page 47) in order to determine the success or failure of `catch/3`. Otherwise, in case the unification fails, the stack keeps unwinding, looking for an earlier invocation of `catch/3`. `Ball` may be any non-variable term.

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`
`Ball` is a variable	`instantiation_error`

If `Ball` does not unify with the `Catcher` argument of any call of `catch/3`, a system error message is displayed and `throw/1` fails.

When `catch/3` calls `Recovery` it uses `call/1` (section 6.2.3, page 47), an `instantiation_error`, a `type_error` or an `existence_error` can then occur depending on `Recovery`.

Portability

ISO control constructs.

7 Prolog built-in predicates

7.1 Type testing

7.1.1 var/1, nonvar/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1, callable/1, list/1, partial_list/1, list_or_partial_list/1

Templates

```
var(?term)                atomic(?term)
nonvar(?term)             compound(?term)
atom(?term)               callable(?term)
integer(?term)            list(?term)
float(?term)              partial_list(?term)
number(?term)             list_or_partial_list(?term)
```

Description

`var(Term)` succeeds if `Term` is currently uninstantiated (which therefore has not been bound to anything, except possibly another uninstantiated variable).

`nonvar(Term)` succeeds if `Term` is currently instantiated (opposite of `var/1`).

`atom(Term)` succeeds if `Term` is currently instantiated to an atom.

`integer(Term)` succeeds if `Term` is currently instantiated to an integer.

`float(Term)` succeeds if `Term` is currently instantiated to a floating point number.

`number(Term)` succeeds if `Term` is currently instantiated to an integer or a floating point number.

`atomic(Term)` succeeds if `Term` is currently instantiated to an atom, an integer or a floating point number.

`compound(Term)` succeeds if `Term` is currently instantiated to a compound term, i.e. a term of arity > 0 (a list or a structure).

`callable(Term)` succeeds if `Term` is currently instantiated to a callable term, i.e. an atom or a compound term.

`list(Term)` succeeds if `Term` is currently instantiated to a list, i.e. the atom `[]` (empty list) or a term with principal functor `'.'/2` and with second argument (the tail) a list.

`partial_list(Term)` succeeds if `Term` is currently instantiated to a partial list, i.e. a variable or a term whose the main functor is `'.'/2` and the second argument (the tail) is a partial list.

`list_or_partial_list(Term)` succeeds if `Term` is currently instantiated to a list or a partial list.

Errors

None.

Portability

`var/1`, `nonvar/1`, `atom/1`, `integer/1`, `float/1`, `number/1`, `atomic/1`, `compound/1` and `callable/1` are ISO predicates.

`list/1`, `partial_list/1` and `list_or_partial_list/1` are GNU Prolog predicates.

7.2 Term unification

7.2.1 `(=)/2` - Prolog unification

Templates

```
=(?term, ?term)
```

Description

`Term1 = Term2` unifies `Term1` and `Term2`. No occurs check is done, i.e. this predicate does not check if a variable is unified with a compound term containing this variable (this can lead to an infinite loop).

`=` is a predefined infix operator (section 7.14.10, page 100).

Errors

None.

Portability

ISO predicate.

7.2.2 `unify_with_occurs_check/2`

Templates

```
unify_with_occurs_check(?term, ?term)
```

Description

`unify_with_occurs_check(Term1, Term2)` unifies `Term1` and `Term2`. The occurs check test is done (i.e. the unification fails if a variable is unified with a compound term containing this variable).

Errors

None.

Portability

ISO predicate.

7.2.3 `(\=)/2` - not Prolog unifiable

Templates

```
\=(?term, ?term)
```

Description

`Term1 \= Term2` succeeds if `Term1` and `Term2` are not unifiable (no occurs check is done).

`\=` is a predefined infix operator (section 7.14.10, page 100).

Errors

None.

Portability

ISO predicate.

7.3 Term comparison

7.3.1 Standard total ordering of terms

The built-in predicates described in this section allows the user to compare Prolog terms. Prolog terms are totally ordered according to the standard total ordering of terms which is as follows (from the smallest term to the greatest):

- variables, oldest first.
- finite domain variables (section 8.1.1, page 165), oldest first.
- floating point numbers, in numeric order.
- integers, in numeric order.
- atoms, in alphabetical (i.e. character code) order.
- compound terms, ordered first by arity, then by the name of the principal functor and by the arguments in left-to-right order.

A list is treated as a compound term (whose principal functor is `'.'/2`).

The portability of the order of variables is not guaranteed (in the ISO reference the oder of variables is system dependent).

7.3.2 (==)/2 - term identical, (\==)/2 - term not identical, (@<)/2 - term less than, (@=<)/2 - term less than or equal to, (@>)/2 - term greater than, (@>=)/2 - term greater than or equal to

Templates

```
==(?term, ?term)
\==(?term, ?term)
@<(?term, ?term)
@=<(?term, ?term)
@>(?term, ?term)
@>=(?term, ?term)
```

Description

These predicates compare two terms according to the standard total ordering of terms (section 7.3.1, page 51).

`Term1 == Term2` succeeds if `Term1` and `Term2` are equal.

`Term1 \== Term2` succeeds if `Term1` and `Term2` are different.

`Term1 @< Term2` succeeds if `Term1` is less than `Term2`.

`Term1 @=< Term2` succeeds if `Term1` is less than or equal to `Term2`.

`Term1 @> Term2` succeeds if `Term1` is greater than `Term2`.

`Term1 @>= Term2` succeeds if `Term1` is greater than or equal to `Term2`.

`==`, `\==`, `@<`, `@=<`, `@>` and `@>=` are predefined infix operators (section 7.14.10, page 100).

Errors

None.

Portability

ISO predicates.

7.3.3 compare/3

Templates

```
compare(?atom, +term, +term)
```

Description

`compare(Result, Term1, Term2)` compares `Term1` and `Term2` according to the standard (section 7.3.1, page 51) and unifies `Result` with:

- the atom `<` if `Term1` is less than `Term2`.
- the atom `=` if `Term1` and `Term2` are equal.
- the atom `>` if `Term1` is greater than `Term2`.

Errors

`Result` is neither a variable nor an atom	`type_error(atom, Result)`

Portability

GNU Prolog predicate.

7.4 Term processing

7.4.1 functor/3

Templates

```
functor(+nonvar, ?atomic, ?integer)
functor(-nonvar, +atomic, +integer)
```

Description

`functor(Term, Name, Arity)` succeeds if the principal functor of `Term` is `Name` and its arity is `Arity`. This predicate can be used in two ways:

- `Term` is not a variable: extract the name (an atom or a number if `Term` is a number) and the arity of `Term` (if `Term` is atomic `Arity` = 0).

- `Term` is a variable: unify `Term` with a general term whose principal functor is given by `Name` and arity is given by `Arity`.

Errors

`Term` and `Name` are both variables	`instantiation_error`
`Term` and `Arity` are both variables	`instantiation_error`
`Term` is a variable and `Name` is neither a variable nor an atomic term	`type_error(atomic, Name)`
`Term` is a variable and `Arity` is neither a variable nor an integer	`type_error(integer, Arity)`
`Term` is a variable, `Name` is a constant but not an atom and `Arity` is an integer > 0	`type_error(atom, Name)`
`Term` is a variable and `Arity` is an integer > `max_arity` flag (section 7.22.1, page 133)	`representation_error(max_arity)`
`Term` is a variable and `Arity` is an integer < 0	`domain_error(not_less_than_zero, Arity)`

Portability

ISO predicate.

7.4.2 `arg/3`

Templates

```
arg(+integer, +compound_term, ?term)
```

Description

`arg(N, Term, Arg)` succeeds if the `N`*th* argument of `Term` is `Arg`.

Errors

`N` is a variable	`instantiation_error`
`Term` is a variable	`instantiation_error`
`N` is neither a variable nor an integer	`type_error(integer, N)`
`Term` is neither a variable nor a compound term	`type_error(compound, Term)`
`N` is an integer < 0	`domain_error(not_less_than_zero, N)`

Portability

ISO predicate.

7.4.3 `(=..)/2` - univ

Templates

```
=..(+nonvar, ?list)
=..(-nonvar, +list)
```

Description

`Term =.. List` succeeds if `List` is a list whose head is the atom corresponding to the principal functor of `Term` and whose tail is a list of the arguments of `Term`.

`=..` is a predefined infix operator (section 7.14.10, page 100).

Errors

`Term` is a variable and `List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
`Term` is a variable and `List` is a list whose head is a variable	`instantiation_error`
`List` is a list whose head `H` is neither an atom nor a variable and whose tail is not the empty list	`type_error(atom, H)`
`List` is a list whose head `H` is a compound term and whose tail is the empty list	`type_error(atomic, H)`
`Term` is a variable and `List` is the empty list	`domain_error(non_empty_list, [])`
`Term` is a variable and the tail of `List` has a length > `max_arity` flag (section 7.22.1, page 133)	`representation_error(max_arity)`

Portability

ISO predicate.

7.4.4 `copy_term/2`

Templates

```
copy_term(?term, ?term)
```

Description

`copy_term(Term1, Term2)` succeeds if `Term2` unifies with a term `T` which is a renamed copy of `Term1`.

Errors

None.

Portability

ISO predicate.

7.4.5 `setarg/4`, `setarg/3`

Templates

```
setarg(+integer, +compound_term, +term, +boolean)
setarg(+integer, +compound_term, +term)
```

Description

`setarg(N, Term, NewValue, Undo)` replaces destructively the `N`*th* argument of `Term` with `NewValue`. This assignment is undone on backtracking if `Undo` = `true`. This should only used if there is no further use of the old value of the replaced argument. If `Undo` = `false` then `NewValue` must be either an atom or an integer.

`setarg(N, Term, NewValue)` is equivalent to `setarg(N, Term, NewValue, true)`.

Errors

`N` is a variable	`instantiation_error`
`N` is neither a variable nor an integer	`type_error(integer, N)`
`N` is an integer < 0	`domain_error(not_less_than_zero, N)`
`Term` is a variable	`instantiation_error`
`Term` is neither a variable nor a compound term	`type_error(compound, Term)`
`NewValue` is neither an atom nor an integer and `Undo` = `false`	`type_error(atomic, NewValue)`
`Undo` is a variable	`instantiation_error`
`Undo` is neither a variable nor a boolean	`type_error(boolean, Undo)`

Portability

GNU Prolog predicate.

7.5 Variable naming/numbering

7.5.1 `name_singleton_vars/1`

Templates

```
name_singleton_vars(?term)
```

Description

`name_singleton_vars(Term)` binds each singleton variable appearing in `Term` with a term of the form `'$VARNAME'('_')`. Such a term can be output by `write_term/3` as a variable name (section 7.14.6, page 96).

Errors

None.

Portability

GNU Prolog predicates.

7.5.2 `name_query_vars/2`

Templates

```
name_query_vars(+list, ?list)
```

Description

`name_query_vars(List, Rest)` for each element of `List` of the form `Name = Var` where `Name` is an atom and `Var` a variable, binds `Var` with the term `'$VARNAME'(Name)`. Such a term can be output by `write_term/3` as a variable name (section 7.14.6, page 96). `Rest` is unified with the list of elements of `List` that have not given rise to a binding. This predicate is provided as a way to name the variable lists obtained returned by `read_term/3` with `variable_names(List)` or `singletons(List)` options (section 7.14.1, page 92).

Errors

`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
`Rest` is neither a partial list nor a list	`type_error(list, Rest)`

Portability

GNU Prolog predicate.

7.5.3 `bind_variables/2`, `numbervars/3`, `numbervars/1`

Templates

```
bind_variables(?term, +var_binding_option_list)
numbervars(?term, +integer, ?integer)
numbervars(?term)
```

Description

`bind_variables(Term, Options)` binds each variable appearing in `Term` according to the options given by `Options`.

Variable binding options: `Options` is a list of variable binding options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `numbervars`: specifies that each variable appearing in `Term` should be bound to a term of the form `'$VAR'(N)` where `N` is an integer. Such a term can be output by `write_term/3` as a variable name (section 7.14.6, page 96). This is the default.
- `namevars`: specifies that each variables appearing in `Term` shall be bound to a term of the form `'$VARNAME'(Name)` where `Name` is the atom that would be output by `write_term/3` seeing a term of the `'$VAR'(N)` where `N` is an integer. Such a term can be output by `write_term/3` as a variable name (section 7.14.6, page 96). This is the alternative to `numbervars`.
- `from(From)`: the first integer `N` to use for number/name variables of `Term` is `From`. The default value is `0`.
- `next(Next)`: when `bind_variables/2` succeeds, `Next` is unified with the (last integer `N`)+1 used to bind the variables of `Term`.
- `exclude(List)`: collects all variable names appearing in `List` to avoid a clash when binding a variable of `Term`. Precisely a number `N` $\geq$ `From` will not be used to bind a variable of `Term` if:
  - there is a sub-term of `List` of the form `'$VAR'(N)` or `'$VARNAME'(Name)` where `Name` is the constant that would be output by `write_term/3` seeing a term of the `'$VAR'(N)`.
  - an element of `List` is of the form `Name = Var` where `Name` is an atom that would be output by `write_term/3` on seeing a term of the from `'$VAR'(N)`. This case allows for lists returned by `read_term/3` (with `variable_names(List)` or `singletons(List)` options) (section 7.14.1, page 92) and by `name_query_vars/2` (section 7.5.2, page 55).

`numbervars(Term, From, Next)` is equivalent to `bind_variables(Term, [from(From), next(Next)]`, i.e. each variable of `Term` is bound to `'$VAR'(N)` where `From` $\leq$ `N` $<$ `Next`.

`numbervars(Term)` is equivalent to `numbervars(Term, 0, _)`.

Errors

`Options` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Options` is neither a partial list nor a list	`type_error(list, Options)`
an element `E` of the `Options` list is neither a variable nor a variable binding option	`domain_error(var_binding_option, E)`
`From` is a variable	`instantiation_error`
`From` is neither a variable nor an integer	`type_error(integer, From)`
`Next` is neither a variable nor an integer	`type_error(integer, Next)`
`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`

Portability

GNU Prolog predicates.

7.5.4 `term_ref/2`

Templates

```
term_ref(+term, ?integer)
term_ref(?term, +integer)
```

Description

`term_ref(Term, Ref)` succeeds if the internal reference of `Term` is `Ref`. This predicate can be used either to obtain the internal reference of a term or to obtain the term associated to a given reference. Note that two identical terms can have different internal references. A good way to use this predicate is to first record the internal reference of a given term and to later re-obtain the term via this reference.

Errors

`Term` and `Ref` are both variables	`instantiation_error`
`Ref` is neither a variable nor an integer	`type_error(integer, Ref)`
`Ref` is an integer < 0	`domain_error(not_less_than_zero, Ref)`

Portability

GNU Prolog predicate.

7.6 Arithmetic

7.6.1 Evaluation of an arithmetic expression

An arithmetic expression is a Prolog term built from numbers, variables, and functors (or operators) that represent arithmetic functions. When an expression is evaluated each variable must be bound to a non-variable expression. An expression evaluates to a number, which may be an integer or a floating point number. The following table details the components of an arithmetic expression, how they are evaluated, the types expected/returned and if they are ISO or an extension:

Expression	Result = *eval*(Expression)	Signature	ISO
Variable	must be bound to a non-variable expression `E`. The result is *eval*(`E`)	IF $\rightarrow$ IF	Y
integer number	this number	I $\rightarrow$ I	Y
floating point number	this number	F $\rightarrow$ F	Y
`+ E`	*eval*(`E`)	IF $\rightarrow$ IF	N
`- E`	- *eval*(`E`)	IF $\rightarrow$ IF	Y
`inc(E)`	*eval*(`E`) + 1	IF $\rightarrow$ IF	N
`dec(E)`	*eval*(`E`) - 1	IF $\rightarrow$ IF	N
`E1 + E2`	*eval*(`E1`) + *eval*(`E2`)	IF, IF $\rightarrow$ IF	Y
`E1 - E2`	*eval*(`E1`) - *eval*(`E2`)	IF, IF $\rightarrow$ IF	Y
`E1 * E2`	*eval*(`E1`) * *eval*(`E2`)	IF, IF $\rightarrow$ IF	Y
`E1 / E2`	*eval*(`E1`) / *eval*(`E2`)	IF, IF $\rightarrow$ F	Y
`E1 // E2`	*rnd*(*eval*(`E1`) / *eval*(`E2`))	I, I $\rightarrow$ I	Y
`E1 rem E2`	*eval*(`E1`) - (*rnd*(*eval*(`E1`) / *eval*(`E2`))**eval*(`E2`))	I, I $\rightarrow$ I	Y
`E1 mod E2`	*eval*(`E1`) - ($\lfloor$*eval*(`E1`) / *eval*(`E2`)$\rfloor$ **eval*(`E2`))	I, I $\rightarrow$ I	Y
`E1 /\ E2`	*eval*(`E1`) bitwise_and *eval*(`E2`)	I, I $\rightarrow$ I	Y
`E1 \/ E2`	*eval*(`E1`) bitwise_or *eval*(`E2`)	I, I $\rightarrow$ I	Y
`E1 ^ E2`	*eval*(`E1`) bitwise_xor *eval*(`E2`)	I, I $\rightarrow$ I	N
`\ E`	bitwise_not *eval*(`E`)	I $\rightarrow$ I	Y
`E1 << E2`	*eval*(`E1`) integer_shift_left *eval*(`E2`)	I, I $\rightarrow$ I	Y
`E1 >> E2`	*eval*(`E1`) integer_shift_right *eval*(`E2`)	I, I $\rightarrow$ I	Y
`abs(E)`	absolute value of *eval*(`E`)	IF $\rightarrow$ IF	Y
`sign(E)`	sign of *eval*(`E`) (-1 if $<$ 0, 0 if = 0, +1 if $>$ 0)	IF $\rightarrow$ IF	Y
`min(E1,E2)`	minimal value between *eval*(`E1`) and *eval*(`E2`)	IF, IF $\rightarrow$?	N
`max(E1,E2)`	maximal value between *eval*(`E1`) and *eval*(`E2`)	IF, IF $\rightarrow$?	N
`E1 ** E2`	*eval*(`E1`) raised to the power of *eval*(`E2`)	IF, IF $\rightarrow$ F	Y
`sqrt(E)`	square root of *eval*(`E`)	IF $\rightarrow$ F	Y
`atan(E)`	arc tangent of *eval*(`E`)	IF $\rightarrow$ F	Y
`cos(E)`	cosine of *eval*(`E`)	IF $\rightarrow$ F	Y
`acos(E)`	arc cosine of *eval*(`E`)	IF $\rightarrow$ F	N
`sin(E)`	sine of *eval*(`E`)	IF $\rightarrow$ F	Y
`asin(E)`	arc sine of *eval*(`E`)	IF $\rightarrow$ F	N
`exp(E)`	*e* raised to the power of *eval*(`E`)	IF $\rightarrow$ F	Y
`log(E)`	natural logarithms of *eval*(`E`)	IF $\rightarrow$ F	Y
`float(E)`	the floating point number equal to *eval*(`E`)	IF $\rightarrow$ F	Y
`ceiling(E)`	rounds *eval*(`E`) upward to the nearest integer	F $\rightarrow$ I	Y
`floor(E)`	rounds *eval*(`E`) downward to the nearest integer	F $\rightarrow$ I	Y
`round(E)`	rounds *eval*(`E`) to the nearest integer	F $\rightarrow$ I	Y
`truncate(E)`	the integer value of *eval*(`E`)	F $\rightarrow$ I	Y
`float_fractional_part(E)`	the float equal to the fractional part of *eval*(`E`)	F $\rightarrow$ F	Y
`float_integer_part(E)`	the float equal to the integer part of *eval*(`E`)	F $\rightarrow$ F	Y

The meaning of the signature field is as follows:

- I $\rightarrow$ I: unary function, the operand must be an integer and the result is an integer.
- F $\rightarrow$ F: unary function, the operand must be a floating point number and the result is a floating point number.
- F $\rightarrow$ I: unary function, the operand must be a floating point number and the result is an integer.
- IF $\rightarrow$ F: unary function, the operand can be an integer or a floating point number and the result is a floating point number.
- IF $\rightarrow$ IF: unary function, the operand can be an integer or a floating point number and the result has the same type as the operand.

- I, I → I: binary function: each operand must be an integer and the result is an integer.
- IF, IF → IF: binary function: each operand can be an integer or a floating point number and the result is a floating point number if at least one operand is a floating point number, an integer otherwise.
- IF, IF → ?: binary function: each operand can be an integer or a floating point number and the result has the same type as the selected operand. This is used for `min` and `max`. Note that in case of equality between an integer and a floating point number the result is an integer.

`is`, `+`, `-`, `*`, `//`, `/`, `rem`, and `mod` are predefined infix operators. `+` and `-` are predefined prefix operators (section 7.14.10, page 100).

Integer division rounding function: the integer division rounding function *rnd*`(X)` rounds the floating point number `X` to an integer. There are two possible definitions (depending on the target machine) for this function which differ on negative numbers:

- *rnd*`(X)` = integer part of `X`, e.g. *rnd*`(-1.5)` = `-1` (round toward 0)
- *rnd*`(X)` = $\lfloor$`X`$\rfloor$, e.g. *rnd*`(-1.5)` = `-2` (round toward $-\infty$)

The definition of this function determines the precise definition of the integer division `(//)/2` and of the integer remainder `(rem)/2`. Rounding toward zero is the most common case. In any case it is possible to test the value (`toward_zero` or `down`) of the `integer_rounding_function` Prolog flag to determine which function being used (section 7.22.1, page 133).

Fast mathematical mode: in order to speed-up integer computations, the GNU Prolog compiler can generate faster code when invoked with the `--fast-math` option (section 3.4.3, page 22). In this mode only integer operations are allowed and a variable in an expression must be bound at evaluation time to an integer. No type checking is done.

Errors

a sub-expression `E` is a variable	`instantiation_error`
a sub-expression `E` is neither a number nor an evaluable functor	`type_error(evaluable, E)`
a sub-expression `E` is a floating point number while an integer is expected	`type_error(integer, E)`
a sub-expression `E` is an integer while a floating point number is expected	`type_error(float, E)`
a division by zero occurs	`evaluation_error(zero_divisor)`

Portability

Refer to the above table to determine which evaluable functors are ISO and which are GNU Prolog extensions. For efficiency reasons, GNU Prolog does not detect the following ISO arithmetic errors: `float_overflow`, `int_overflow, int_underflow`, and `undefined`.

7.6.2 `(is)/2` - evaluate expression

Templates

```
is(?nonvar, +evaluable)
```

Description

`Result is Expression` succeeds if `Result` can be unified with *eval*(`Expression`). Refer to the evaluation of an arithmetic expression for the definition of the *eval* function (section 7.6.1, page 57).

`is` is a predefined infix operator (section 7.14.10, page 100).

Errors

Refer to the evaluation of an arithmetic expression for possible errors (section 7.6.1, page 57).

Portability

ISO predicate.

7.6.3 (`=:=`)/2 - arithmetic equal, (`=\=`)/2 - arithmetic not equal, (`<`)/2 - arithmetic less than, (`=<`)/2 - arithmetic less than or equal to, (`>`)/2 - arithmetic greater than, (`>=`)/2 - arithmetic greater than or equal to

Templates

```
=:=(+evaluable, +evaluable)          =<(+evaluable, +evaluable)
=\=(+evaluable, +evaluable)          >(+evaluable, +evaluable)
<(+evaluable, +evaluable)            >=(+evaluable, +evaluable)
```

Description

`Expr1 =:= Expr2` succeeds if $eval(\texttt{Expr1}) = eval(\texttt{Expr2})$.

`Expr1 =\= Expr2` succeeds if $eval(\texttt{Expr1}) \neq eval(\texttt{Expr2})$.

`Expr1 < Expr2` succeeds if $eval(\texttt{Expr1}) < eval(\texttt{Expr2})$.

`Expr1 =< Expr2` succeeds if $eval(\texttt{Expr1}) \leq eval(\texttt{Expr2})$.

`Expr1 > Expr2` succeeds if $eval(\texttt{Expr1}) > eval(\texttt{Expr2})$.

`Expr1 >= Expr2` succeeds if $eval(\texttt{Expr1}) \geq eval(\texttt{Expr2})$.

Refer to the evaluation of an arithmetic expression for the definition of the *eval* function (section 7.6.1, page 57).

`=:=`, `=\=`, `<`, `=<`, `>` and `>=` are predefined infix operators (section 7.14.10, page 100).

Errors

Refer to the evaluation of an arithmetic expression for possible errors (section 7.6.1, page 57).

Portability

ISO predicates.

7.7 Dynamic clause management

7.7.1 Introduction

Static and dynamic procedures: a procedure is either dynamic or static. All built-in predicates are static. A user-defined procedure is static by default unless a `dynamic/1` directive precedes its definition (section 6.1.2, page 41). Adding a clause to a non-existent procedure creates a dynamic procedure. The clauses of a dynamic procedure can be altered (e.g. using `asserta/1`), the clauses of a static procedure cannot be altered.

Private and public procedures: each procedure is either public or private. A dynamic procedure is always public. Each built-in predicate is private, and a static user-defined procedure is private by default unless a `public/1` directive precedes its definition (section 6.1.3, page 41). If a dynamic declaration exists it is unnecessary to add a public declaration since a dynamic procedure is also public. A clause of a public procedure can be inspected (e.g. using `clause/2`), a clause of a private procedure cannot be inspected.

A logical database update view: any change in the database that occurs as the result of executing a goal (e.g. when a sub-goal is a call of `assertz/1` or `retract/1`) only affects subsequent activations. The change does not affect any activation that is currently being executed. Thus the database is frozen during the execution of a goal, and the list of clauses defining a predication is fixed at the moment of its execution.

7.7.2 `asserta/1`, `assertz/1`

Templates

```
asserta(+clause)
assertz(+clause)
```

Description

`asserta(Clause)` first converts the term `Clause` to a clause and then adds it to the current internal database. The predicate concerned must be dynamic (section 7.7.1, page 60) or undefined and the clause is inserted before the first clause of the predicate. If the predicated is undefined it is created as a dynamic procedure.

`assertz(Clause)` acts like `asserta/1` except that the clause is added at the end of all existing clauses of the concerned predicate.

Converting a term `Clause` to a clause `Clause1`:

- extract the head and the body of `Clause`: either `Clause` = `(Head :- Body)` or `Clause` = `Head` and `Body` = `true`.
- `Head` must be a callable term (or else the conversion fails).
- convert `Body` to a body clause (i.e. a goal) `Body1`.
- the converted clause `Clause1` = `(Head :- Body1)`.

Converting a term `T` to a goal:

- if `T` is a variable it is replaced by the term `call(T)`.
- if `T` is a control construct `(',')/2`, `(;)/2` or `(->)/2` each argument of the control construct is recursively converted to a goal.
- if `T` is a callable term it remains unchanged.
- otherwise the conversion fails (`T` is neither a variable nor a callable term).

Errors

`Head` is a variable	`instantiation_error`
`Head` is neither a variable nor a callable term	`type_error(callable, Head)`
`Body` cannot be converted to a goal	`type_error(callable, Body)`
The predicate indicator `Pred` of `Head` is that of a static procedure	`permission_error(modify, static_procedure, Pred)`

Portability

ISO predicates.

7.7.3 `retract/1`

Templates

```
retract(+clause)
```

Description

`retract(Clause)` erases the first clause of the database that unifies with `Clause`. The concerned predicate must be a dynamic procedure (section 7.7.1, page 60). Removing all clauses of a procedure does not erase the procedure definition. To achieve this use `abolish/1` (section 7.7.6, page 63). `retract/1` is re-executable on backtracking.

Errors

`Head` is a variable	`instantiation_error`
`Head` is neither a variable nor a callable term	`type_error(callable, Head)`
The predicate indicator `Pred` of `Head` is that of a static procedure	`permission_error(modify, static_procedure, Pred)`

Portability

ISO predicate. In the ISO reference, the operation associated to the `permission_error` is `access` while it is `modify` in GNU Prolog. This seems to be an error of the ISO reference since for `asserta/1` (which is similar in spirit to `retract/1`) the operation is also `modify`.

7.7.4 `retractall/1`

Templates

```
retractall(+head)
```

Description

`retractall(Head)` erases all clauses whose head unifies with `Head`. The concerned predicate must be a dynamic procedure (section 7.7.1, page 60). The procedure definition is not removed so that it is found by `current_predicate/1` (section 7.8.1, page 64). `abolish/1` should be used to remove the procedure (section 7.7.6, page 63).

Errors

`Head` is a variable	`instantiation_error`
`Head` is not a callable term	`type_error(callable, Head)`
The predicate indicator `Pred` of `Head` is that of a static procedure	`permission_error(modify, static_procedure, Pred)`

Portability

GNU Prolog predicate.

7.7.5 clause/2

Templates

```
clause(+head, ?callable_term)
```

Description

`clause(Head, Body)` succeeds if there exists a clause in the database that unifies with `Head :- Body`. The predicate in question must be a public procedure (section 7.7.1, page 60). Clauses are delivered from the first to the last. This predicate is re-executable on backtracking.

Errors

`Head` is a variable	`instantiation_error`
`Head` is neither a variable nor a callable term	`type_error(callable, Head)`
The predicate indicator `Pred` of `Head` is that of a private procedure	`permission_error(access, private_procedure, Pred)`
`Body` is neither a variable nor a callable term	`type_error(callable, Body)`

Portability

ISO predicate.

7.7.6 abolish/1

Templates

```
abolish(+predicate_indicator)
```

Description

`abolish(Pred)` removes from the database the procedure whose predicate indicator is `Pred`. The concerned predicate must be a dynamic procedure (section 7.7.1, page 60).

Errors

`Pred` is a variable	`instantiation_error`
`Pred` is a term `Name/Arity` and either `Name` or `Arity` is a variable	`instantiation_error`
`Pred` is neither a variable nor a predicate indicator	`type_error(predicate_indicator, Pred)`
`Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer	`type_error(integer, Arity)`
`Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom	`type_error(atom, Name)`
`Pred` is a term `Name/Arity` and `Arity` is an integer <0	`domain_error(not_less_than_zero, Arity)`
`Pred` is a term `Name/Arity` and `Arity` is an integer $>$ `max_arity` flag (section 7.22.1, page 133)	`representation_error(max_arity)`
The predicate indicator `Pred` is that of a static procedure	`permission_error(modify, static_procedure, Pred)`

Portability

ISO predicate.

7.8 Predicate information

7.8.1 `current_predicate/1`

Templates

`current_predicate(?predicate_indicator)`

Description

`current_predicate(Pred)` succeeds if there exists a predicate indicator of a defined procedure that unifies with `Pred`. All user defined procedures are found, whether static or dynamic. Internal system procedures whose name begins with `'$'` are not found. A user-defined procedure is found even when it has no clauses. A user-defined procedure is not found if it has been abolished. To conform to the ISO reference, built-in predicates are not found except if the `strict_iso` Prolog flag is switched off (section 7.22.1, page 133). This predicate is re-executable on backtracking.

Errors

`Pred` is neither a variable nor a predicate indicator	`type_error(predicate_indicator, Pred)`
`Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer	`type_error(integer, Arity)`
`Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom	`type_error(atom, Name)`
`Pred` is a term `Name/Arity` and `Arity` is an integer <0	`domain_error(not_less_than_zero, Arity)`
`Pred` is a term `Name/Arity` and `Arity` is an integer $>$ `max_arity` flag (section 7.22.1, page 133)	`representation_error(max_arity)`

Portability

ISO predicate.

7.8.2 predicate_property/2

Templates

```
predicate_property(?predicate_indicator, ?predicate_property)
```

Description

`predicate_property(Pred, Property)` succeeds if there exists a predicate indicator of a defined procedure that unifies with `Pred` and if `Property` unifies with one of the properties of the procedure. All user defined procedures and built-in predicates are found. Internal system procedures whose name begins with `'$'` are not found. This predicate is re-executable on backtracking.

Predicate properties:

- `static`: if the procedure is static.
- `dynamic`: if the procedure is dynamic.
- `private`: if the procedure is private.
- `public`: if the procedure is public.
- `user`: if the procedure is a user-defined procedure.
- `built_in`: if the procedure is a Prolog built-in predicate.
- `built_in_fd`: if the procedure is an FD built-in predicate.
- `native_code`: if the procedure is compiled in native code.
- `prolog_file(File)`: source file from which the predicate has been read.
- `prolog_line(Line)`: line number of the source file.

Errors

`Pred` is neither a variable nor a predicate indicator	`type_error(predicate_indicator, Pred)`
`Pred` is a term `Name/Arity` and `Arity` is neither a variable nor an integer	`type_error(integer, Arity)`
`Pred` is a term `Name/Arity` and `Name` is neither a variable nor an atom	`type_error(atom, Name)`
`Pred` is a term `Name/Arity` and `Arity` is an integer < 0	`domain_error(not_less_than_zero, Arity)`
`Pred` is a term `Name/Arity` and `Arity` is an integer $>$ `max_arity` flag (section 7.22.1, page 133)	`representation_error(max_arity)`
`Property` is neither a variable nor a predicate property term	`domain_error(predicate_property, Property)`
`Property` = `prolog_file(File)` and `File` is neither a variable nor an atom	`type_error(atom, File)`
`Property` = `prolog_line(Line)` and `Line` is neither a variable nor an integer	`type_error(integer, Line)`

Portability

GNU Prolog predicate.

7.9 All solutions

7.9.1 Introduction

It is sometimes useful to collect all solutions for a goal. This can be done by repeatedly backtracking and gradually building the list of solutions. The following built-in predicates are provided to automate this process.

The built-in predicates described in this section invoke `call/1` (section 6.2.3, page 47) on the argument `Goal`. When efficiency is crucial and `Goal` is complex it is better to define an auxiliary predicate which can then be compiled, and have `Goal` call this predicate.

7.9.2 `findall/3`

Templates

```
findall(?term, +callable_term, ?list)
```

Description

`findall(Template, Goal, Instances)` succeeds if `Instances` unifies with the list of values to which a variable `X` not occurring in `Template` or `Goal` would be instantiated by successive re-executions of `call(Goal), X = Template` after systematic replacement of all variables in `X` by new variables. Thus, the order of the list `Instances` corresponds to the order in which the proofs are found.

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`
`Instances` is neither a partial list nor a list	`type_error(list, Instances)`

Portability

ISO predicate.

7.9.3 `bagof/3`, `setof/3`

Templates

```
bagof(?term, +callable_term, ?list)
setof(?term, +callable_term, ?list)
```

Description

`bagof(Template, Goal, Instances)` assembles as a list the set of solutions of `Goal` for each different instantiation of the free variables in `Goal`. The elements of each list are in order of solution, but the order in which each list is found is undefined. This predicate is re-executable on backtracking.

Free variable set: `bagof/3` groups the solutions of `Goal` according to the free variables in `Goal`. This set corresponds to all variables occurring in `Goal` but not in `Template`. It is sometimes useful to exclude some additional variables of `Goal`. For that, `bagof/3` recognizes a goal of the form `T^Goal` and exclude all variables occuring in `T` from the free variable set. `(^)/2` can be viewed as an *existential quantifier* (the logical reading of `X^Goal` being "there exists an `X` such that `Goal` is true"). The use of this existential qualifier is superfluous outside `bagof/3` (and `setof/3`) and then is not recognized.

`(^)/2` is a predefined infix operator (section 7.14.10, page 100).

`setof(Template, Goal, Instances)` is equivalent to `bagof(Template,Goal,I), sort(I,Instances)`. Each list is then a sorted list (duplicate elements are removed).

From the implementation point of view `setof/3` is as fast as `bagof/3`. Both predicates use an in-place (i.e. destructive) sort (section 7.20.12, page 125) and require the same amount of memory.

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`
`Instances` is neither a partial list nor a list	`type_error(list, Instances)`

Portability

ISO predicates.

7.10 Streams

7.10.1 Introduction

A stream provides a logical view of a source/sink.

Sources and sinks: a program can output results to a sink or input data from a source. A source/sink may be a file (regular file, terminal, device,...), a constant term, a pipe, a socket,...

Associating a stream to a source/sink: to manipulate a source/sink it must be associated to a stream. This provides a logical and uniform view of the source/sink whatever its type. Once this association has been established, i.e. a stream has been created, all subsequent references to the source/sink are made by referring the stream. A stream is unidirectional: it is either an input stream or an output stream. For a classical file, the association is done by opening the file (whose name is specified as an atom) with the `open/4` (section 7.10.6, page 70). GNU Prolog makes it possible to treat a Prolog constant term as a source/sink and provides built-in predicates to associate a stream to such a term (section 7.11, page 82). GNU Prolog provides operating system interface predicates defining pipes between GNU Prolog and child processes with streams associated to these pipes, e.g. `popen/3` (section 7.27.21, page 153). Similarly, socket interface predicates associate streams to a socket to allow the communication, e.g. `socket_connect/4` (section 7.28.5, page 159).

Stream-term: a stream-term identifies a stream during a call of an input/output built-in predicate. It is created as a result of associating a stream to a source/sink (section above). A stream-term is a compound term of the form `'$stream'(I)` where `I` is an integer.

Stream aliases: any stream may be associated with a stream alias which is an atom which may be used to refer to that stream. The association can be done at open time or using `add_stream_alias/2` (section 7.10.20, page 79). Such an association automatically ends when the stream is closed. A particular alias only refers to at most one stream at any one time. However, more than one alias can be associated to a stream. Most built-in predicates which have a stream-term as an input argument also accept a stream alias as that argument. However, built-in predicates which return a stream-term do not accept a stream alias.

Standard streams: two streams are predefined and open during the execution of every goal: the standard input stream which has the alias `user_input` and the standard output stream which has the alias `user_output`. A goal which attempts to close either standard stream succeeds, but does not close the stream.

Current streams: during execution there is a current input stream and a current output stream. By default, the current input and output streams are the standard input and output streams, but the built-in predicates `set_input/1` (section 7.10.4, page 69) and `set_output/1` (section 7.10.5, page 70) can be used to change them. When the current input stream is closed, the standard input stream becomes the current input stream. When the current output stream is closed, the standard output stream becomes the current output stream.

Text streams and binary streams: a text stream is a sequence of characters. A text stream is also regarded as a sequence of lines where each line is a possibly empty sequence of characters followed by a new line character. GNU Prolog may add or remove space characters at the ends of lines in order to conform to the conventions for representing text streams in the operating system. A binary stream is a sequence of bytes. Only a few built-in predicates can deal with binary streams, e.g. `get_byte/2` (section 7.13, page 89).

Stream positions: the stream position of a stream identifies an absolute position of the source/sink to which the stream is connected and defines where in the source/sink the next input or output will take place. A stream position is a ground term of the form `'$stream_position'(I1, I2, I3, I4)` where `I1`, `I2`, `I3` and `I4` are integers. Stream positions are used to reposition a stream (when possible) using for instance `set_stream_position/2` (section 7.10.13, page 75).

The position end of stream: when all data of a stream *S* has been input *S* has a stream position end-of-stream. At this stream position a goal to input more data will return a specific value to indicate that end of stream has been reached (e.g. `-1` for `get_code/2` or `end_of_file` for `get_char/2`,...). When this terminating value has been input, the stream has a stream position past-end-of-stream.

Buffering mode: input/output on a stream can be buffered (line-buffered or block-buffered) or not buffered at all. The buffering mode can be specified at open time or using `set_stream_buffering/2` (section 7.10.27, page 82). Line buffering is used on output streams, output data are only written to the sink when a new-line character is output (or at the close time). Block buffering is used on input or output. On input streams, when an input is requested on the source, if the buffer is empty, all available characters are read (within the limits of the size of the buffer), subsequent reads will first use the characters in the buffer. On output streams, output data are stored in the buffer and only when the buffer is full is it physically written on the sink. Thus, an output to a buffered stream may not be sent immediately to the sink connected to that stream. When it is necessary to be certain that output has been delivered, the built-in predicate `flush_output/1` (section 7.10.8, page 72) should be used. Finally, it is also possible to use non-buffered streams, in that case input/output are directly done on the connected source/sink. This can be useful for communication purposes (e.g. sockets) or when a precise control is needed, e.g. `select/5` (section 7.27.25, page 155).

Stream mirrors: any stream may be associated with mirror streams specified at open time or using `add_stream_mirror/2` (section 7.10.22, page 79). Then, all characters/bytes read from/written to the stream are also written on each mirror stream. The association automatically ends when either the stream or the mirror stream is closed. It is also possible to explicitely remove a mirror stream using `remove_stream_mirror/2` (section 7.10.23, page 80).

7.10.2 `current_input/1`

Templates

`current_input(?stream)`

Description

`current_input(Stream)` unifies `Stream` with the stream-term identifying the current input stream.

Errors

`Stream` is neither a variable nor a stream	`domain_error(stream, Stream)`

Portability

ISO predicate.

7.10.3 `current_output/1`

Templates

`current_output(?stream)`

Description

`current_output(Stream)` unifies `Stream` with the stream-term identifying the current output stream.

Errors

`Stream` is neither a variable nor a stream	`domain_error(stream, Stream)`

Portability

ISO predicate.

7.10.4 `set_input/1`

Templates

`set_input(+stream_or_alias)`

Description

`set_input(SorA)` sets the current input stream to be the stream associated with the stream-term or alias `SorA`.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`

Portability

ISO predicate.

7.10.5 set_output/1

Templates

```
set_output(+stream_or_alias)
```

Description

`set_output(SorA)` sets the current output stream to be the stream associated with the stream-term or alias `SorA`.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an input stream	`permission_error(output, stream, SorA)`

Portability

ISO predicate.

7.10.6 open/4, open/3

Templates

```
open(+source_sink, +io_mode, -stream, +stream_option_list)
open(+source_sink, +io_mode, -stream)
```

Description

`open(SourceSink, Mode, Stream, Options)` opens the source/sink `SourceSink` for input or output as indicated by `Mode` and the list of stream-options `Options` and unifies `Stream` with the stream-term which is associated to this stream. See `absolute_file_name/2` for information about the syntax of `SourceSink` (section 7.26.1, page 141).

Input/output modes: `Mode` is an atom which defines the input/output operations that may be performed the stream. Possible modes are:

- `read`: the source/sink is a source and must already exist. Input starts at the beginning of the source.
- `write`: the source/sink is a sink. If the sink already exists then it is emptied else an empty sink is created. Output starts at the beginning of that sink.
- `append`: the source/sink is a sink. If the sink does not exist it is created. Output starts at the end of that sink.

Stream options: `Options` is a list of stream options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `type(text/binary)`: specifies whether the stream is a text stream or a binary stream. The default value is `text`.

- `reposition(true/false)`: specifies whether it is possible to reposition the stream. The default value is `true` except if the stream cannot be repositioned (e.g. a terminal).
- `eof_action(error/eof_code/reset)`: specifies the effect of attempting to input from a stream whose stream position is past-end-of-stream:
  - `error`: a `permission_error` is raised signifying that no more input exists in this stream.
  - `eof_code`: the result of input is as if the stream position is end-of-stream.
  - `reset`: the stream position is reset so that it is not past-end-of-stream, and another attempt is made to input from it (this is useful when inputting from a terminal).

  The default value is `eof_code`.
- `alias(Alias)`: specifies that the atom `Alias` is to be an alias for the stream. By default no alias is attached to the stream. Several aliases can be defined for a same stream.
- `mirror(Mirror)`: specifies the stream associated with the stream-term or alias `Mirror` is a mirror for the stream. By default no mirro is attached to the stream. Several mirrors can be defined for a same stream.
- `buffering(none/line/block)`: specifies which type of buffering is used by input/output operations on this stream:
  - `none`: no buffering.
  - `line`: output operations buffer data emitted until a new-line occurs
  - `block`: input/output operations buffer data until a given number (implementation dependant) of characters/bytes have been treated.

  The default value is `line` for a terminal (TTY), `block` otherwise.

`open(SourceSink, Mode, Stream, Options)` is equivalent to `open(SourceSink, Mode, Stream, [])`.

Errors

`SourceSink` is a variable	`instantiation_error`
`Mode` is a variable	`instantiation_error`
`Options` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Mode` is neither a variable nor an atom	`type_error(atom, Mode)`
`Options` is neither a partial list nor a list	`type_error(list, Options)`
`Stream` is not a variable	`type_error(variable, Stream)`
`SourceSink` is neither a variable nor a source/sink	`domain_error(source_sink, SourceSink)`
`Mode` is an atom but not an input/output mode	`domain_error(io_mode, Mode)`
an element `E` of the `Options` list is neither a variable nor a stream-option	`domain_error(stream_option, E)`
the source/sink specified by `SourceSink` does not exist	`existence_error(source_sink, SourceSink)`
the source/sink specified by `SourceSink` cannot be opened	`permission_error(open, source_sink, SourceSink)`
an element `E` of the `Options` list is `alias(A)` and `A` is already associated with an open stream	`permission_error(open, source_sink, alias(A))`
an element `E` of the `Options` list is `mirror(M)` and `M` is not associated with an open stream	`existence_error(stream, M)`
an element `E` of the `Options` list is `mirror(M)` and `M` iis an input stream	`permission_error(output, stream, M)`
an element `E` of the `Options` list is `reposition(true)` and it is not possible to reposition this stream	`permission_error(open, source_sink, reposition(true))`

Portability

ISO predicates. The `mirror/1` and `buffering/1` stream options are GNU Prolog extensions.

7.10.7 close/2, close/1

Templates

```
close(+stream_or_alias, +close_option_list)
close(+stream_or_alias)
```

Description

`close(SorA, Options)` closes the stream associated with the stream-term or alias `SorA`. If `SorA` is the standard input stream or the standard output stream `close/2` simply succeeds else the associated source/sink is physically closed. If `SorA` is the current input stream the current input stream becomes the standard input stream `user_input`. If `SorA` is the current output stream the current output stream becomes the standard output stream `user_output`.

Close options: `Options` is a list of close options. For the moment only one option is available:

- `force(true/false)`: with `false`, if an error occurs when trying to close the source/sink, the stream is not closed and an error (`system_error` or `resource_error`) is raised (but `close/2` succeeds). With `true`, if an error occurs it is ignored and the stream is closed. The purpose of `force/1` option is to allow an error handling routine to do its best to reclaim resources. The default value is `false`.

`close(SorA)` is equivalent to `close(SorA, [])`.

Errors

`SorA` is a variable	`instantiation_error`
`Options` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Options` is neither a partial list nor a list	`type_error(list, Options)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
an element `E` of the `Options` list is neither a variable nor a close-option	`domain_error(close_option, E)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` needs a special close (section 7.11, page 82)	`system_error(needs_special_close)`

Portability

ISO predicates. The `system_error(needs_special_close)` is a GNU Prolog extension.

7.10.8 flush_output/1, flush_output/0

Templates

```
flush_output(+stream_or_alias)
flush_output
```

Description

`flush_output(SorA)` sends any buffered output characters/bytes to the stream.

`flush_output/0` applies to the current output stream.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an input stream	`permission_error(output, stream, SorA)`

Portability

ISO predicates.

7.10.9 `current_stream/1`

Templates

`current_stream(?stream)`

Description

`current_stream(Stream)` succeeds if there exists a stream-term that unifies with `Stream`. This predicate is re-executable on backtracking.

Errors

`Stream` is neither a variable nor a stream-term	`domain_error(stream, Stream)`

Portability

GNU Prolog predicate.

7.10.10 `stream_property/2`

Templates

`stream_property(?stream, ?stream_property)`

Description

`stream_property(Stream, Property)` succeeds if `current_stream(Stream)` succeeds (section 7.10.9, page 73) and if `Property` unifies with one of the properties of the stream. This predicate is re-executable on backtracking.

Stream properties:

- `file_name(F)`: the name of the connected source/sink.
- `mode(M)`: `M` is the open mode (`read`, `write`, `append`).
- `input`: if it is an input stream.
- `output`: if it is an output stream.
- `alias(A)`: `A` is an alias of the stream.
- `mirror(M)`: `M` is a mirror stream of the stream.
- `type(T)`: `T` is the type of the stream (`text`, `binary`).
- `reposition(R)`: `R` is the reposition boolean (`true`, `false`).

- `eof_action(A)`: `A` is the end-of-file action (`error`, `eof_code`, `reset`).
- `buffering(B)`: `B` is the buffering mode (`none`, `line`, `block`).
- `end_of_stream(E)`: `E` is the current end-of-stream status (`not`, `at`, `past`). If the stream position is end-of-stream then `E` is unified with `at` else if the stream position is past-end-of-stream then `E` is unified with `past` else `E` is unified with `not`.
- `position(P)`: `P` is the stream-position term associated to the current position.

Errors

`Stream` is a variable	`instantiation_error`
`Stream` is neither a variable nor a stream-term	`domain_error(stream, Stream)`
`Property` is neither a variable nor a stream property	`domain_error(stream_property, Property)`
`Property` = `file_name(E)`, `mode(E)`, `alias(E)`, `end_of_stream(E)`, `eof_action(E)`, `reposition(E)`, `type(E)` or `buffering(E)` and `E` is neither a variable nor an atom	`type_error(atom, E)`

Portability

ISO predicate. The `buffering/1` property is a GNU Prolog extension.

7.10.11 `at_end_of_stream/1`, `at_end_of_stream/0`

Templates

```
at_end_of_stream(+stream_or_alias)
at_end_of_stream
```

Description

`at_end_of_stream(SorA)` succeeds if the stream associated with stream-term or alias `SorA` has a stream position end-of-stream or past-end-of-stream. This predicate can be defined using `stream_property/2` (section 7.10.10, page 73).

`at_end_of_stream/0` applies to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`

Portability

ISO predicates. The `permission_error(input, stream, SorA)` is a GNU Prolog extension.

7.10.12 `stream_position/2`

Templates

`stream_position(+stream_or_alias, ?stream_position)`

Description

`stream_position(SorA, Position)` succeeds unifying `Position` with the stream-position term associated to the current position of the stream-term or alias `SorA`. This predicate can be defined using `stream_property/2` (section 7.10.10, page 73).

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Position` is neither a variable nor a stream-position term	`domain_error(stream_position, Position)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`

Portability

GNU Prolog predicate.

7.10.13 `set_stream_position/2`

Templates

`set_stream_position(+stream_or_alias, +stream_position)`

Description

`set_stream_position(SorA, Position)` sets the position of the stream associated with the stream-term or alias `SorA` to `Position`. `Position` should have previously been returned by `stream_property/2` (section 7.10.10, page 73) or by `stream_position/2` (section 7.10.12, page 74).

Errors

`SorA` is a variable	`instantiation_error`
`Position` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Position` is neither a variable nor a stream-position term	`domain_error(stream_position, Position)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` has stream property `reposition(false)`	`permission_error(reposition, stream, SorA)`

Portability

ISO predicate.

7.10.14 `seek/4`

Templates

`seek(+stream_or_alias, +stream_seek_method, +integer, ?integer)`

Description

`seek(SorA, Whence, Offset, NewOffset)` sets the position of the stream associated with the stream-term or alias `SorA` to `Offset` according to `Whence` and unifies `NewOffset` with the new offset from the beginning of the file. `seek/4` can only be used on binary streams. `Whence` is an atom from:

- `bof`: the position is set relatively to the begin of the file (`Offset` should be ≥ 0).
- `current`: the position is set relatively to the current position (`Offset` can be ≥ 0 or ≤ 0).
- `eof`: the position is set relatively to the end of the file (`Offset` should be ≤ 0).

This predicate is an interface to the C Unix function `lseek(2)`.

Errors

`SorA` is a variable	`instantiation_error`
`Whence` is a variable	`instantiation_error`
`Offset` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Whence` is neither a variable nor an atom	`type_error(atom, Whence)`
`Whence` is an atom but not a valid stream seek method	`domain_error(stream_seek_method, Whence)`
`Offset` is neither a variable nor an integer	`type_error(integer, Offset)`
`NewOffset` is neither a variable nor an integer	`type_error(integer, NewOffset)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` has stream property `reposition(false)`	`permission_error(reposition, stream, SorA)`
`SorA` is associated with a text stream	`permission_error(reposition, text_stream, SorA)`

Portability

GNU Prolog predicate.

7.10.15 `character_count/2`

Templates

```
character_count(+stream_or_alias, ?integer)
```

Description

`character_count(SorA, Count)` unifies `Count` with the number of characters/bytes read/written on the stream associated with stream-term or alias `SorA`.

Errors

`SorA` is a variable	`instantiation_error`
`Count` is neither a variable nor an integer	`type_error(integer, Count)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`

Portability

GNU Prolog predicate.

7.10.16 `line_count/2`

Templates

`line_count(+stream_or_alias, ?integer)`

Description

`line_count(SorA, Count)` unifies `Count` with the number of lines read/written on the stream associated with the stream-term or alias `SorA`. This predicate can only be used on text streams.

Errors

`SorA` is a variable	`instantiation_error`
`Count` is neither a variable nor an integer	`type_error(integer, Count)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(access, binary_stream, SorA)`

Portability

GNU Prolog predicate.

7.10.17 `line_position/2`

Templates

`line_position(+stream_or_alias, ?integer)`

Description

`line_position(SorA, Count)` unifies `Count` with the number of characters read/written on the current line of the stream associated with the stream-term or alias `SorA`. This predicate can only be used on text streams.

Errors

`SorA` is a variable	`instantiation_error`
`Count` is neither a variable nor an integer	`type_error(integer, Count)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(access, binary_stream, SorA)`

Portability

GNU Prolog predicate.

7.10.18 `stream_line_column/3`

Templates

`stream_line_column(+stream_or_alias, ?integer, ?integer)`

Description

`stream_line_column(SorA, Line, Column)` unifies `Line` (resp. `Column`) with the current line number (resp. column number) of the stream associated with the stream-term or alias `SorA`. This predicate can only be used on text streams. Note that `Line` corresponds to the value returned by `line_count/2` + 1 (section 7.10.16, page 77) and `Column` to the value returned by `line_position/2` + 1 (section 7.10.17, page 77).

Errors

`SorA` is a variable	`instantiation_error`
`Line` is neither a variable nor an integer	`type_error(integer, Line)`
`Column` is neither a variable nor an integer	`type_error(integer, Column)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(access, binary_stream, SorA)`

Portability

GNU Prolog predicate.

7.10.19 `set_stream_line_column/3`

Templates

`set_stream_line_column(+stream_or_alias, +integer, +integer)`

Description

`set_stream_line_column(SorA, Line, Column)` sets the stream position of the stream associated with the stream-term or alias `SorA` according to the line number `Line` and the column number `Column`. This predicate can only be used on text streams. It first repositions the stream to the beginning of the file and then reads character by character until the required position is reached.

Errors

`SorA` is a variable	`instantiation_error`
`Line` is a variable	`instantiation_error`
`Column` is a variable	`instantiation_error`
`Line` is neither a variable nor an integer	`type_error(integer, Line)`
`Column` is neither a variable nor an integer	`type_error(integer, Column)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(reposition, binary_stream, SorA)`
`SorA` has stream property `reposition(false)`	`permission_error(reposition, stream, SorA)`

Portability

GNU Prolog predicate.

7.10.20 `add_stream_alias/2`

Templates

    add_stream_alias(+stream_or_alias, +atom)

Description

`add_stream_alias(SorA, Alias)` adds `Alias` as a new alias to the stream associated with the stream-term or alias `SorA`.

Errors

`SorA` is a variable	`instantiation_error`
`Alias` is a variable	`instantiation_error`
`Alias` is neither a variable nor an atom	`type_error(atom, Alias)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`Alias` is already associated with an open stream	`permission_error(add_alias, source_sink, alias(Alias))`

Portability

GNU Prolog predicate.

7.10.21 `current_alias/2`

Templates

    current_alias(?stream, ?atom)

Description

`current_alias(Stream, Alias)` succeeds if `current_stream(Stream)` succeeds (section 7.10.9, page 73) and if `Alias` unifies with one of the aliases of the stream. It can be defined using `stream_property/2` (section 7.10.10, page 73). This predicate is re-executable on backtracking.

Errors

`Stream` is neither a variable nor a stream-term	`domain_error(stream, Stream)`
`Alias` is neither a variable nor an atom	`type_error(atom, Alias)`

Portability

GNU Prolog predicate.

7.10.22 `add_stream_mirror/2`

Templates

    add_stream_mirror(+stream_or_alias, +stream_or_alias)

Description

`add_stream_mirror(SorA, Mirror)` adds the stream associated with the stream-term or alias `Mirror` as a new mirror to the stream associated with the stream-term or alias `SorA`. After this, all characters (or bytes) read from (or written to) `SorA` are also written to `Mirror`. This mirroring occurs until `Mirror` is explicitely removed using `remove_stream_mirror/2` (section 7.10.23, page 80) or implicitely when `Mirror` is closed. Several mirror streams can be associated to a same stream. If `Mirror` represents the same stream as `SorA` or if `Mirror` is already a mirror for `SorA`, no mirror is added.

Errors

`SorA` is a variable	`instantiation_error`
`Mirror` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Mirror` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, Mirror)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`Mirror` is not associated with an open stream	`existence_error(stream, Mirror)`
`Mirror` is an input stream	`permission_error(output, stream, Mirror)`

Portability

GNU Prolog predicate.

7.10.23 `remove_stream_mirror/2`

Templates

```
remove_stream_mirror(+stream_or_alias, +stream_or_alias)
```

Description

`remove_stream_mirror(SorA, Mirror)` removes the stream associated with the stream-term or alias `Mirror` from the list of mirrors of the stream associated with the stream-term or alias `SorA`. This predicate fails if `Mirror` is not a mirror stream for `SorA`.

Errors

`SorA` is a variable	`instantiation_error`
`Mirror` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Mirror` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, Mirror)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`Mirror` is not associated with an open stream	`existence_error(stream, Mirror)`

Portability

GNU Prolog predicate.

7.10.24 `current_mirror/2`

Templates

```
current_mirror(?stream, ?stream)
```

Description

`current_mirror(Stream, M)` succeeds if `current_stream(Stream)` succeeds (section 7.10.9, page 73) and if `M` unifies with one of the mirrors of the stream. It can be defined using `stream_property/2` (section 7.10.10, page 73). This predicate is re-executable on backtracking.

Errors

`Stream` is neither a variable nor a stream-term	`domain_error(stream, Stream)`
`M` is neither a variable nor a stream-term	`domain_error(stream, M)`

Portability

GNU Prolog predicate.

7.10.25 `set_stream_type/2`

Templates

```
set_stream_type(+stream_or_alias, +atom)
```

Description

`set_stream_type(SorA, Type)` updates the type associated with stream-term or alias `SorA`. The value of `Type` is an atom in `text` or `binary` as for `open/4` (section 7.10.6, page 70). The type of a stream can only be changed before any input/output operation is executed.

Errors

`SorA` is a variable	`instantiation_error`
`Type` is a variable	`instantiation_error`
`Type` is neither a variable nor a valid type	`domain_error(stream_type, Type)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
An I/O operation has already been executed on `SorA`	`permission_error(modify, stream, SorA)`

Portability

GNU Prolog predicate.

7.10.26 `set_stream_eof_action/2`

Templates

```
set_stream_eof_action(+stream_or_alias, +atom)
```

Description

`set_stream_eof_action(SorA, Action)` updates the `eof_action` option associated with the stream-term or alias `SorA`. The value of `Action` is one of the atoms `error`, `eof_code`, `reset` as for `open/4` (section 7.10.6, page 70).

Errors

SorA is a variable	instantiation_error
Action is a variable	instantiation_error
Action is neither a variable nor a valid eof action	domain_error(eof_action, Action)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an output stream	permission_error(modify, stream, SorA)

Portability

GNU Prolog predicate.

7.10.27 set_stream_buffering/2

Templates

```
set_stream_buffering(+stream_or_alias, +atom)
```

Description

`set_stream_buffering(SorA, Buffering)` updates the buffering mode associated with the stream-term or alias `SorA`. The value of `Buffering` is one of the atoms `none`, `line` or `block` as for `open/4` (section 7.10.6, page 70). This predicate may only be used after opening a stream and before any other operations have been performed on it.

Errors

SorA is a variable	instantiation_error
Buffering is a variable	instantiation_error
Buffering is neither a variable nor a valid buffering mode	domain_error(buffering_mode, Buffering)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)

Portability

GNU Prolog predicate.

7.11 Constant term streams

7.11.1 Introduction

Constant term streams allow the user to consider a constant term (atom, character list or character code list) as a source/sink by associating to them a stream. Reading from a constant term stream will deliver the characters of the constant term as if they had been read from a standard file. Characters written on a constant term stream are stored to form the final constant term when the stream is closed. The built-in predicates described in this section allow the user to open and close a constant term stream for input or output. However, very often, a constant term stream is created to be only read or written once and then closed. To avoid the creation and the destruction of such a stream, GNU Prolog offers several built-in predicates to perform single input/output from/to constant terms (section 7.15, page 104).

7.11.2 open_input_atom_stream/2, open_input_chars_stream/2, open_input_codes_stream/2

Templates

```
open_input_atom_stream(+atom, -stream)
open_input_chars_stream(+character_list, -stream)
open_input_codes_stream(+character_code_list, -stream)
```

Description

`open_input_atom_stream(Atom, Stream)` unifies `Stream` with the stream-term which is associated to a new input text-stream whose data are the characters of `Atom`.

`open_input_chars_stream(Chars, Stream)` is similar to `open_input_atom_stream/2` except that data are the content of the character list `Chars`.

`open_input_codes_stream(Codes, Stream)` is similar to `open_input_atom_stream/2` except that data are the content of the character code list `Codes`.

Errors

`Stream` is not a variable	`type_error(variable, Stream)`
`Atom` is a variable	`instantiation_error`
`Chars` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Codes` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Atom` is neither a variable nor a an atom	`type_error(atom, Atom)`
`Chars` is neither a partial list nor a list	`type_error(list, Chars)`
`Codes` is neither a partial list nor a list	`type_error(list, Codes)`
an element `E` of the `Chars` list is neither a variable nor a character	`type_error(character, E)`
an element `E` of the `Codes` list is neither a variable nor an integer	`type_error(integer, E)`
an element `E` of the `Codes` list is an integer but not a character code	`representation_error(character_code)`

Portability

GNU Prolog predicates.

7.11.3 close_input_atom_stream/1, close_input_chars_stream/1, close_input_codes_stream/1

Templates

```
close_input_atom_stream(+stream_or_alias)
close_input_chars_stream(+stream_or_alias)
close_input_codes_stream(+stream_or_alias)
```

Description

`close_input_atom_stream(SorA)` closes the constant term stream associated with the stream-term or alias `SorA`. `SorA` must a stream open with `open_input_atom_stream/2` (section 7.11.1, page 82).

`close_input_chars_stream(SorA)` acts similarly for a character list stream.

`close_input_codes_stream(SorA)` acts similarly for a character code list stream.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(close, stream, SorA)`
`SorA` is a stream-term or alias but does not refer to a constant term stream.	`domain_error(term_stream_or_alias, SorA)`

Portability

GNU Prolog predicates.

7.11.4 `open_output_atom_stream/1`, `open_output_chars_stream/1`, `open_output_codes_stream/1`

Templates

```
open_output_atom_stream(-stream)
open_output_chars_stream(-stream)
open_output_codes_stream(-stream)
```

Description

`open_output_atom_stream(Stream)` unifies `Stream` with the stream-term which is associated to a new output text-stream. All characters written to this stream are collected and will be returned as an atom when the stream is closed by `close_ouput_atom_stream/2` (section 7.11.5, page 84).

`open_output_chars_stream(Stream)` is similar to `open_output_atom_stream/1` except that the result will be a character list.

`open_output_codes_stream(Stream)` is similar to `open_output_atom_stream/1` except that the result will be a character code list.

Errors

`Stream` is not a variable	`type_error(variable, Stream)`

Portability

GNU Prolog predicates.

7.11.5 `close_output_atom_stream/2`, `close_output_chars_stream/2`, `close_output_codes_stream/2`

Templates

```
close_output_atom_stream(+stream_or_alias, ?atom)
close_output_chars_stream(+stream_or_alias, ?character_list)
close_output_codes_stream(+stream_or_alias, ?character_code_list)
```

Description

`close_output_atom_stream(SorA, Atom)` closes the constant term stream associated with the stream-term or alias `SorA`. `SorA` must be associated to a stream open with `open_output_atom_stream/1` (section 7.11.4, page 84). `Atom` is unified with an atom formed with all characters written on the stream.

`close_output_chars_stream(SorA, Chars)` acts similarly for a character list stream.

`close_output_codes_stream(SorA, Codes)` acts similarly for a character code list stream.

Errors

`SorA` is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Chars` is neither a partial list nor a list	`type_error(list, Chars)`
`Codes` is neither a partial list nor a list	`type_error(list, Codes)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an input stream	`permission_error(close, stream, SorA)`
`SorA` is a stream-term or alias but does not refer to a constant term stream	`domain_error(term_stream_or_alias, SorA)`

Portability

GNU Prolog predicates.

7.12 Character input/output

These built-in predicates enable a single character or character code to be input from and output to a text stream. The atom `end_of_file` is returned as character to indicate the end-of-file. `-1` is returned as character code to indicate the end-of-file.

7.12.1 `get_char/2`, `get_char/1`, `get_code/1`, `get_code/2`

Templates

```
get_char(+stream_or_alias, ?in_character)
get_char(?in_character)
get_code(+stream_or_alias, ?in_character_code)
get_code(?in_character_code)
```

Description

`get_char(SorA, Char)` succeeds if `Char` unifies with the next character read from the stream associated with the stream-term or alias `SorA`.

`get_code/2` is similar to `get_char/2` but deals with character codes.

`get_char/1` and `get_code/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Char` is neither a variable nor an in-character	`type_error(in_character, Char)`
`Code` is neither a variable nor an integer	`type_error(integer, Code)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`
The entity input from the stream is not a character	`representation_error(character)`
`Code` is an integer but not an in-character code	`representation_error(in_character_code)`

Portability

ISO predicates.

7.12.2 `get_key/2`, `get_key/1` `get_key_no_echo/2`, `get_key_no_echo/1`

Templates

```
get_key(+stream_or_alias, ?integer)
get_key(?integer)
get_key_no_echo(+stream_or_alias, ?integer)
get_key_no_echo(?integer)
```

Description

`get_key(Code, SorA)` succeeds if `Code` unifies with the character code of the next key read from the stream associated with the stream-term or alias `SorA`. It is intended to read a single key from the keyboard (thus `SorA` should refer to current input stream). No buffering is performed (a character is read as soon as available) and function keys can also be read (in that case, `Code` is an integer > 255). The read character is echoed if it is printable.

This facility is only possible if the `linedit` facility has been installed (section 3.2.5, page 18) otherwise `get_key/2` behaves similarly to `get_code/2` (section 7.12.1, page 85) (the code of the first character is returned) but also pumps remaining characters until a character < space (0x20) is read (in particular RETURN). The same behavior occurs if `SorA` does not refer to the current input stream or if this stream is not attached to a terminal.

`get_key_no_echo/2` behaves similarly to `get_key/2` except that the read character is not echoed.

`get_key/1` and `get_key_no_echo/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Code` is neither a variable nor an integer	`type_error(integer, Code)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`

Portability

GNU Prolog predicates.

7.12.3 `peek_char/2`, `peek_char/1`, `peek_code/1`, `peek_code/2`

Templates

```
peek_char(+stream_or_alias, ?in_character)
peek_char(?in_character)
peek_code(+stream_or_alias, ?in_character_code)
peek_code(?in_character_code)
```

Description

`peek_char(SorA, Char)` succeeds if `Char` unifies with the next character that will be read from the stream associated with the stream-term or alias `SorA`. The character is not read.

`peek_code/2` is similar to `peek_char/2` but deals with character codes.

`peek_char/1` and `peek_code/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Char` is neither a variable nor an in-character	`type_error(in_character, Char)`
`Code` is neither a variable nor an integer	`type_error(integer, Code)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`
The entity input from the stream is not a character	`representation_error(character)`
`Code` is an integer but not an in-character code	`representation_error(in_character_code)`

Portability

ISO predicates.

7.12.4 unget_char/2, unget_char/1, unget_code/2, unget_code/1

Templates

```
unget_char(+stream_or_alias, +character)
unget_char(+character)
unget_code(+stream_or_alias, +character_code)
unget_code(+character_code)
```

Description

`unget_char(SorA, Char)` pushes back `Char` onto the stream associated with the stream-term or alias `SorA`. `Char` will be the next character read by `get_char/2`. The maximum number of characters that can be cumulatively pushed back is given by the `max_unget` Prolog flag (section 7.22.1, page 133).

`unget_code/2` is similar to `unget_char/2` but deals with character codes.

`unget_char/1` and `unget_code/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Char` is a variable	`instantiation_error`
`Code` is a variable	`instantiation_error`
`Char` is neither a variable nor a character	`type_error(character, Char)`
`Code` is neither a variable nor an integer	`type_error(integer, Code)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`Code` is an integer but not a character code	`representation_error(character_code)`

Portability

GNU Prolog predicates.

7.12.5 put_char/2, put_char/1, put_code/1, put_code/2, nl/1, nl/0

Templates

```
put_char(+stream_or_alias, +character)
put_char(+character)
put_code(+stream_or_alias, +character_code)
put_code(+character_code)
nl(+stream_or_alias)
nl
```

Description

`put_char(SorA, Char)` writes `Char` onto the stream associated with the stream-term or alias `SorA`.

`put_code/2` is similar to `put_char/2` but deals with character codes.

`nl(SorA)` writes a new-line character onto the stream associated with the stream-term or alias `SorA`. This is equivalent to `put_char(SorA, '\n')`.

put_char/1, put_code/1 and nl/0 apply to the current output stream.

Errors

SorA is a variable	instantiation_error
Char is a variable	instantiation_error
Code is a variable	instantiation_error
Char is neither a variable nor a character	type_error(character, Char)
Code is neither a variable nor an integer	type_error(integer, Code)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an input stream	permission_error(output, stream, SorA)
SorA is associated with a binary stream	permission_error(output, binary_stream, SorA)
Code is an integer but not a character code	representation_error(character_code)

Portability

ISO predicates.

7.13 Byte input/output

These built-in predicates enable a single byte to be input from and output to a binary stream. -1 is returned to indicate the end-of-file.

7.13.1 get_byte/2, get_byte/1

Templates

```
get_byte(+stream_or_alias, ?in_byte)
get_byte(?in_byte)
```

Description

get_byte(SorA, Byte) succeeds if Byte unifies with the next byte read from the stream associated with the stream-term or alias SorA.

get_byte/1 applies to the current input stream.

Errors

SorA is a variable	instantiation_error
Byte is neither a variable nor an in-byte	type_error(in_byte, Byte)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an output stream	permission_error(input, stream, SorA)
SorA is associated with a text stream	permission_error(input, text_stream, SorA)
SorA has stream properties end_of_stream(past) and eof_action(error)	permission_error(input, past_end_of_stream, SorA)

Portability

ISO predicates.

7.13.2 peek_byte/2, peek_byte/1

Templates

```
peek_byte(+stream_or_alias, ?in_byte)
peek_byte(?in_byte)
```

Description

`peek_byte(SorA, Byte)` succeeds if `Byte` unifies with the next byte that will be read from the stream associated with the stream-term or alias `SorA`. The byte is not read.

`peek_byte/1` applies to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Byte` is neither a variable nor an in-byte	`type_error(in_byte, Byte)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a text stream	`permission_error(input, text_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`

Portability

ISO predicates.

7.13.3 unget_byte/2, unget_byte/1

Templates

```
unget_byte(+stream_or_alias, +byte)
unget_byte(+byte)
```

Description

`unget_byte(SorA, Byte)` pushes back `Byte` onto the stream associated with the stream-term or alias `SorA`. `Byte` will be the next byte read by `get_byte/2`. The maximum number of bytes that can be successively pushed back is given by the `max_unget` Prolog flag (section 7.22.1, page 133).

`unget_byte/1` applies to the current input stream.

Errors

SorA is a variable	instantiation_error
Byte is a variable	instantiation_error
Byte is neither a variable nor a byte	type_error(byte, Byte)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an output stream	permission_error(input, stream, SorA)
SorA is associated with a text stream	permission_error(input, text_stream, SorA)

Portability

GNU Prolog predicates.

7.13.4 put_byte/2, put_byte/1

Templates

```
put_byte(+stream_or_alias, +byte)
put_byte(+byte)
```

Description

put_byte(SorA, Byte) writes Byte onto the stream associated with the stream-term or alias SorA.

put_byte/1 applies to the current output stream.

Errors

SorA is a variable	instantiation_error
Byte is a variable	instantiation_error
Byte is neither a variable nor a byte	type_error(byte, Byte)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an output stream	permission_error(output, stream, SorA)
SorA is associated with a text stream	permission_error(output, text_stream, SorA)

Portability

GNU Prolog predicates.

7.14 Term input/output

These built-in predicates enable a Prolog term to be input from or output to a text stream. The atom end_of_file is returned as term to indicate the end-of-file. The syntax of such terms can also be altered by changing the operators (section 7.14.10, page 100), and making some characters equivalent to others (section 7.14.12, page 102) if the char_conversion Prolog flag is on (section 7.22.1, page 133). Double quoted tokens will be returned as an atom or a character list or a character code list depending on the value of the double_quotes Prolog flag (section 7.22.1, page 133). Similarly, back quoted tokens are returned depending on the value of the back_quotes Prolog flag.

7.14.1 read_term/3, read_term/2, read/2, read/1

Templates

```
read_term(+stream_or_alias, ?term, +read_option_list)
read_term(?term, +read_option_list)
read(+stream_or_alias, ?term)
read(?term)
```

Description

`read_term(SorA, Term, Options)` is true if `Term` unifies with the next term read from the stream associated with the stream-term or alias `SorA` according to the options given by `Options`.

Read options: `Options` is a list of read options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `variables(VL)`: `VL` is unified with the list of all variables of the input term, in left-to-right traversal order. Anonymous variables are included in the list `VL`.
- `variable_names(VNL)`: `VNL` is unified with the list of pairs `Name = Var` where `Var` is a named variable of the term and `Name` is the atom associated to the name of `Var`. Anonymous variables are not included in the list `VNL`.
- `singletons(SL)`: `SL` is unified with the list of pairs `Name = Var` where `Var` is a named variable which occurs only once in the term and `Name` is the atom associated to the name of `Var`. Anonymous variables are not included in the list `SL`.
- `syntax_error(error/warning/fail)`: specifies the effect of a syntax error:
  - `error`: a `syntax_error` is raised.
  - `warning`: a warning message is displayed and the predicate fails.
  - `fail`: the predicate quietly fails.

  The default value is the value of the `syntax_error` Prolog flag (section 7.22.1, page 133).
- `end_of_term(dot/eof)`: specifies the end-of-term delimiter: `dot` is the classical full-stop delimiter (a dot followed with a layout character), `eof` is the end-of-file delimiter. This option is useful for predicates like `read_term_from_atom/3` (section 7.15.1, page 104) to avoid to add a terminal dot at the end of the atom. The default value is `dot`.

`read(SorA, Term)` is equivalent to `read_term(SorA, Term, [])`.

`read_term/2` and `read/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Options` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`Options` is neither a partial list nor a list	`type_error(list, Options)`
an element `E` of the `Options` list is neither a variable nor a valid read option	`domain_error(read_option, E)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`
a syntax error occurs and the value of the `syntax_error` Prolog flag is `error` (section 7.22.1, page 133)	`syntax_error(`*atom explaining the error*`)`

Portability

ISO predicates. The ISO reference raises a `representation_error(Flag)` where `Flag` is `max_arity`, `max_integer`, or `min_integer` when the read term breaches an implementation defined limit specified by `Flag`. GNU Prolog detects neither `min_integer` nor `max_integer` violation and treats a `max_arity` violation as a syntax error. The read options `syntax_error/1` and `end_of_term/1` are GNU Prolog extensions.

7.14.2 `read_atom/2`, `read_atom/1`, `read_integer/2`, `read_integer/1`, `read_number/2`, `read_number/1`

Templates

```
read_atom(+stream_or_alias, ?atom)
read_atom(?atom)
read_integer(+stream_or_alias, ?integer)
read_integer(?integer)
read_number(+stream_or_alias, ?number)
read_number(?number)
```

Description

`read_atom(SorA, Atom)` succeeds if `Atom` unifies with the next atom read from the stream associated with the stream-term or alias `SorA`.

`read_integer(SorA, Integer)` succeeds if `Integer` unifies with the next integer read from the stream associated with the stream-term or alias `SorA`.

`read_number(SorA, Number)` succeeds if `Number` unifies with the next number (integer or floating point number) read from the stream associated with the stream-term or alias `SorA`.

`read_atom/1`, `read_integer/1` and `read_number/1` apply to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Integer` is neither a variable nor an integer	`type_error(integer, Integer)`
`Number` is neither a variable nor a number	`type_error(number, Number)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`
a syntax error occurs and the value of the `syntax_error` Prolog flag is `error` (section 7.22.1, page 133)	`syntax_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.14.3 `read_token/2`, `read_token/1`

Templates

```
read_token(+stream_or_alias, ?nonvar)
read_token(?nonvar)
```

Description

`read_token(SorA, Token)` succeeds if `Token` unifies with the encoding of the next Prolog token read from the stream associated with stream-term or alias `SorA`.

Token encoding:

- `var(A)`: a variable is read whose name is the atom `A`.
- an atom `A`: an atom `A` is read.
- integer `N`: an integer `N` is read.
- floating point number `N`: a floating point number `N` is read.
- `string(A)`: a string (double quoted item) is read whose characters forms the atom `A`.
- `punct(P)`: a punctuation character `P` is read (`P` is a one-character atom in `()[]{|}`, the atom `full_stop` or the atom `end_of_file`).
- `back_quotes(A)`: a back quoted item is read whose characters forms the atom `A`.
- `extended(A)`: an extended character `A` (an atom) is read.

As for `read_term/3`, the behavior of `read_token/2` can be affected by some Prolog flags (section 7.14, page 91).

`read_token/1` applies to the current input stream.

Errors

`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an output stream	`permission_error(input, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(input, binary_stream, SorA)`
`SorA` has stream properties `end_of_stream(past)` and `eof_action(error)`	`permission_error(input, past_end_of_stream, SorA)`
a syntax error occurs and the value of the `syntax_error` Prolog flag is `error` (section 7.22.1, page 133)	`syntax_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.14.4 `syntax_error_info/4`

Templates

```
syntax_error_info(?atom, ?integer, ?integer, ?atom)
```

Description

`syntax_error_info(FileName, Line, Column, Error)` returns the information associated to the last syntax error. `Line` is the line number of the error, `Column` is the column number of the error and `Error` is an atom explaining the error.

Errors

`FileName` is neither a variable nor an atom	`type_error(atom, FileName)`
`Line` is neither a variable nor an integer	`type_error(integer, Line)`
`Column` is neither a variable nor an integer	`type_error(integer, Column)`
`Error` is neither a variable nor an atom	`type_error(atom, Error)`

Portability

GNU Prolog predicate.

7.14.5 `last_read_start_line_column/2`

Templates

```
last_read_start_line_column(?integer, ?integer)
```

Description

`last_read_start_line_column(Line, Column)` unifies `Line` and `Column` with the line number and the column number associated to the start of the last read predicate. This predicate can be used after calling one of the following predicates: `read_term/3`, `read_term/2`, `read/2`, `read/1` (section 7.14.1, page 92), `read_atom/2`, `read_atom/1`, `read_integer/2`, `read_integer/1`, `read_number/2`, `read_number/1` (section 7.14.2, page 93) or `read_token/2`, `read_token/1` (section 7.14.3, page 94).

Errors

`Line` is neither a variable nor an integer	`type_error(integer, Line)`
`Column` is neither a variable nor an integer	`type_error(integer, Column)`

Portability

GNU Prolog predicate.

7.14.6 write_term/3, write_term/2, write/2, write/1, writeq/2, writeq/1, write_canonical/2, write_canonical/1, display/2, display/1, print/2, print/1

Templates

```
write_term(+stream_or_alias, ?term, +write_option_list)
write_term(?term, +write_option_list)
write(+stream_or_alias, ?term)
write(?term)
writeq(+stream_or_alias, ?term)
writeq(?term)
write_canonical(+stream_or_alias, ?term)
write_canonical(?term)
display(+stream_or_alias, ?term)
display(?term)
print(+stream_or_alias, ?term)
print(?term)
```

Description

`write_term(SorA, Term, Options)` writes `Term` to the stream associated with the stream-term or alias `SorA` according to the options given by `Options`.

Write options: `Options` is a list of write options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `quoted(true/false)`: if `true` each atom and functor is quoted if this would be necessary for the term to be input by `read_term/3`. If `false` no extra quotes are written. The default value is `false`.
- `ignore_ops(true/false)`: if `true` each compound term is output in functional notation (neither operator notation nor list notation is used). If `false` operator and list notations are used. The default value is `false`.
- `numbervars(true/false)`: if `true` a term of the form `'$VAR'(N)`, where `N` is an integer, is output as a variable name (see below). If `false` such a term is output normally (according to the other options). The default value is `true`.
- `namevars(true/false)`: if `true` a term of the form `'$VARNAME'(Name)`, where `Name` is an atom, is output as a variable name (see below). If `false` such a term is output normally (according to the other options). The default value is `true`.
- `space_args(true/false)`: if `true` an extra space character is emitted after each comma separating the arguments of a compound term in functional notation or of a list. If `false` no extra space is emitted. The default value is `false`.
- `portrayed(true/false)`: if `true` and if there exists a predicate `portray/1`, `write_term/3` acts as follows: if `Term` is a variable it is simply written. If `Term` is non-variable then it is passed to `portray/1`. If this succeeds then it is assumed that `Term` has been output. Otherwise `write_term/3` outputs the principal functor of `Term` (`Term` itself if it is atomic) according to other options and recursively calls `portray/1` on the components of `Term` (if it is a compound term). With

ignore_ops(false) a list is first passed to portray/1 and only if this call fails each element of the list is passed to portray/1 (thus every sub-list is not passed). The default value is false.

- max_depth(N): controls the depth of output for compound terms. N is an integer specifying the depth. The output of a term whose depth is greater than N gives rise to the output of ... (3 dots). By default there is no depth limit.
- priority(N): specifies the starting priority to output the term. This option controls if Term should be enclosed in brackets. N is a positive integer ≤ 1200. By default N = 1200.

Variable numbering: when the numbervars(true) option is passed to write_term/3 any term of the form '$VAR'(N) where N is an integer is output as a variable name consisting of a capital letter possibly followed by an integer. The capital letter is the (I+1)*th* letter of the alphabet and the integer is J, where I = N mod 26 and J = N // 26. The integer J is omitted if it is zero. For example:

```
'$VAR'(0)  is written as A
'$VAR'(1)  is written as B
...
'$VAR'(25) is written as Z
'$VAR'(26) is written as A1
'$VAR'(27) is written as B1
```

Variable naming: when the namevars(true) option is passed to write_term/3 any term of the form '$VARNAME'(Name) where Name is an atom is output as a variable name consisting of the characters Name. For example: '$VARNAME'('A') is written as A (even in the presence of the quoted(true) option).

write(SorA, Term) is equivalent to write_term(SorA, Term, []).

writeq(SorA, Term) is equivalent to write_term(SorA, Term, [quoted(true)]).

write_canonical(SorA, Term) is equivalent to write_term(SorA, Term, [quoted(true), ignore_ops(true), numbervars(false)]).

display(SorA, Term) is equivalent to write_term(SorA, Term, [ignore_ops(true), numbervars(false)]).

print(SorA, Term) is equivalent to write_term(SorA, Term, [numbervars(false), portrayed(true)]).

write_term/2, write/1, writeq/1, write_canonical/1, display/1 and print/1 apply to the current output stream.

Errors

SorA is a variable	instantiation_error
Options is a partial list or a list with an element E which is a variable	instantiation_error
Options is neither a partial list nor a list	type_error(list, Options)
SorA is neither a variable nor a stream-term or alias	domain_error(stream_or_alias, SorA)
an element E of the Options list is neither a variable nor a valid write-option	domain_error(write_option, E)
SorA is not associated with an open stream	existence_error(stream, SorA)
SorA is an input stream	permission_error(output, stream, SorA)
SorA is associated with a binary stream	permission_error(output, binary_stream, SorA)

Portability

ISO predicates except `display/1-2` and `print/1-2` that are GNU Prolog predicates. `namevars/1`, `space_args/1`, `portrayed/1`, `max_depth/1` and `priority/1` options are GNU Prolog extensions.

7.14.7 `format/3`, `format/2`

Templates

```
format(+stream_or_alias, +character_code_list_or_atom, +list)
format(+character_code_list_or_atom, +list)
```

Description

`format(SorA, Format, Arguments)` writes the `Format` string replacing each format control sequence `F` by the corresponding element of `Arguments` (formatted according to `F`) to the stream associated with the stream-term or alias `SorA`.

Format control sequences: the general format of a control sequence is `'~NC'`. The character `C` determines the type of the control sequence. `N` is an optional numeric argument. An alternative form of `N` is `'*'`. `'*'` implies that the next argument `Arg` in `Arguments` should be used as a numeric argument in the control sequence. The use of C `printf()` formatting sequence (beginning by the character `%`) is also allowed. The following control sequences are available:

Format sequence	type of the argument	Description
`~Na`	atom	print the atom without quoting. `N` is minimal number of characters to print using spaces on the rigth if needed (default: the length of the atom)
`~Nc`	character code	print the character associated to the code. `N` is the number of times to print the character (default: 1)
`~Nf` `~Ne` `~NE` `~Ng` `~NG`	float expression	pass the argument `Arg` and `N` to the C `printf()` function as: if `N` is not specified `printf("%f",Arg)` else `printf("%.Nf",Arg)`. Similarly for `~Ne`, `~NE`, `~Ng` and `~NG`
`~Nd`	integer expression	print the argument. `N` is the number of digits after the decimal point. If `N` is 0 no decimal point is printed (default: 0)
`~ND`	integer expression	identical to `~Nd` except that `','` separates groups of three digits to the left of the decimal point
`~Nr`	integer expression	print the argument according to the radix `N`. $2 \leq$ `N` ≤ 36 (default: 8). The letters `a-z` denote digits > 9
`~NR`	integer expression	identical to `~Nr` except that the letters `A-Z` denote digits > 9
`~Ns`	character code list	print exactly `N` characters (default: the length of the list)
`~NS`	character list	print exactly `N` characters (default: the length of the list)
`~i`	term	ignore the current argument
`~k`	term	pass the argument to `write_canonical/1` (section 7.14.6, page 96)
`~p`	term	pass the argument to `print/1` (section 7.14.6, page 96)
`~q`	term	pass the argument to `writeq/1` (section 7.14.6, page 96)
`~w`	term	pass the argument to `write/1` (section 7.14.6, page 96)
`~~`	none	print the character `'~'`
`~Nn`	none	print `N` new-line characters (default: 1)
`~N`	none	print a new-line character if not at the beginning of a line
`~?`	atom	use the argument as a nested format string
`%F`	atom, integer or float expression	interface to the C function `printf(3)` for outputting atoms (C string), integers and floating point numbers. `*` are also allowed.

`format/2` applies to the current output stream.

Errors

`SorA` is a variable	`instantiation_error`
`Format` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Arguments` is a partial list	`instantiation_error`
`Format` is neither a partial list nor a list or an atom	`type_error(list, Format)`
`Arguments` is neither a partial list nor a list	`type_error(list, Arguments)`
an element `E` of the `Format` list is neither a variable nor a character code	`representation_error(character_code, E)`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
an element `E` of Format is not a valid format control sequence	`domain_error(format_control_sequence, E)`
the `Arguments` list does not contain sufficient elements	`domain_error(non_empty_list, [])`
an element `E` of the `Arguments` list is a variable while a non-variable term was expected	`instantiation_error`
an element `E` of the `Arguments` list is neither variable nor an atom while an atom was expected	`type_error(atom, E)`
an element `E` of the `Arguments` cannot be evaluated as an arithmetic expression while an integer or a floating point number was expected	an arithmetic error (section 7.6.1, page 57)
an element `E` of the `Arguments` list is neither variable nor character code while a character code was expected	`representation_error(character_code, E)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an input stream	`permission_error(output, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(output, binary_stream, SorA)`

Portability

GNU Prolog predicates.

7.14.8 `portray_clause/2`, `portray_clause/1`

Templates

```
portray_clause(+stream_or_alias, +clause)
portray_clause(+clause)
```

Description

`portray_clause(SorA, Clause)` pretty prints `Clause` to the stream associated with the stream-term or alias `SorA`. `portray_clause/2` uses the variable binding predicates `name_singleton_vars/1` (section 7.5.1, page 55) and `numbervars/1` (section 7.5.3, page 56). This predicate is used by `listing/1` (section 7.23.3, page 138).

`portray_clause/1` applies to the current output stream.

Errors

`Clause` is a variable	`instantiation_error`
`Clause` is neither a variable nor a callable term	`type_error(callable, Clause)`
`SorA` is a variable	`instantiation_error`
`SorA` is neither a variable nor a stream-term or alias	`domain_error(stream_or_alias, SorA)`
`SorA` is not associated with an open stream	`existence_error(stream, SorA)`
`SorA` is an input stream	`permission_error(output, stream, SorA)`
`SorA` is associated with a binary stream	`permission_error(output, binary_stream, SorA)`

Portability

GNU Prolog predicates.

7.14.9 `get_print_stream/1`

Templates

```
get_print_stream(?stream)
```

Description

`get_print_stream(Stream)` unifies `Stream` with the stream-term associated to the output stream used by `print/2` (section 7.14.6, page 96). The purpose of this predicate is to allow a user-defined `portray/1` predicate to identify the output stream in use.

Errors

`Stream` is neither a variable nor a stream-term	`domain_error(stream, Stream)`

Portability

GNU Prolog predicate.

7.14.10 `op/3`

Templates

```
op(+integer, +operator_specifier, +atom_or_atom_list)
```

Description

`op(Priority, OpSpecifier, Operator)` alters the operator table. `Operator` is declared as an operator with properties defined by specifier `OpSpecifier` and `Priority`. `Priority` must be an integer ≥ 0 and ≤ 1200. If `Priority` is 0 then the operator properties of `Operator` (if any) are canceled. `Operator` may also be a list of atoms in which case all of them are declared to be operators. In general, operators can be removed from the operator table and their priority or specifier can be changed. However, it is an error to attempt to change the `','` operator from its initial status. An atom can have multiple operator definitions (e.g. prefix and infix like `+`) however an atom cannot have both an infix and a postfix operator definitions.

Operator specifiers: the following specifiers are available:

Specifier	Type	Associativity
`fx`	prefix	no
`fy`	prefix	yes
`xf`	postfix	no
`yf`	postfix	yes
`xfx`	infix	no
`yfx`	infix	left
`xfy`	infix	right

Prolog predefined operators:

Priority	Specifier	Operators
1200	`xfx`	`:-  -->`
1200	`fx`	`:-`
1100	`xfy`	`;`
1050	`xfy`	`->`
1000	`xfy`	`,`
900	`fy`	`\+`
700	`xfx`	`=  \=  =..  ==  \==  @<  @=<  @>  @>=  is  =:=  =\=  <  =<  >  >=`
600	`xfy`	`:`
500	`yfx`	`+  -  /\  \/`
400	`yfx`	`*  /  //  rem  mod  <<  >>`
200	`xfy`	`**  ^`
200	`fy`	`+  -  \`

FD predefined operators:

Priority	Specifier	Operators
750	`xfy`	`#<=>  #\<=>`
740	`xfy`	`#==>  #\==>`
730	`xfy`	`##  #\/  #\\/`
720	`yfx`	`#/\  #\/\`
710	`fy`	`#\`
700	`xfx`	`#=  #\=  #<  #=<  #>  #>=  #=#  #\=#  #<#  #=<#  #>#  #>=#`
500	`yfx`	`+  -`
400	`yfx`	`*  /  //  rem`
200	`xfy`	`**`
200	`fy`	`+  -`

Errors

`Priority` is a variable	`instantiation_error`
`OpSpecifier` is a variable	`instantiation_error`
`Operator` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Priority` is neither a variable nor an integer	`type_error(integer, Priority)`
`OpSpecifier` is neither a variable nor an atom	`type_error(atom, OpSpecifier)`
`Operator` is neither a partial list nor a list nor an atom	`type_error(list, Operator)`
an element `E` of the `Operator` list is neither a variable nor an atom	`type_error(atom, E)`
`Priority` is an integer not ≥ 0 and ≤ 1200	`domain_error(operator_priority, Priority)`
`OpSpecifier` is not a valid operator specifier	`domain_error(operator_specifier, OpSpecifier)`
`Operator` is `','` or an element of the `Operator` list is `','`	`permission_error(modify, operator, ',')`
`OpSpecifier` is a specifier such that `Operator` would have a postfix and an infix definition.	`permission_error(create, operator, Operator)`

Portability

ISO predicate.

The ISO reference implies that if a program calls `current_op/3`, then modifies an operator definition by calling `op/3` and backtracks into the call to `current_op/3`, then the changes are guaranteed not to affect that `current_op/3` goal. This is not guaranteed by GNU Prolog.

7.14.11 `current_op/3`

Templates

```
current_op(?integer, ?operator_specifier, ?atom)
```

Description

`current_op(Priority, OpSpecifier, Operator)` succeeds if `Operator` is an operator with properties defined by specifier `OpSpecifier` and `Priority`. This predicate is re-executable on backtracking.

Errors

`Priority` is neither a variable nor an operator priority	`domain_error(operator_priority, Priority)`
`OpSpecifier` is neither a variable nor an operator specifier	`domain_error(operator_specifier, OpSpecifier)`
`Operator` is neither a variable nor an atom	`type_error(atom, Operator)`

Portability

ISO predicate.

7.14.12 `char_conversion/2`

Templates

```
char_conversion(+character, +character)
```

Description

`char_conversion(InChar, OutChar)` alters the character-conversion mapping. This mapping is used by the following read predicates: `read_term/3` (section 7.14.1, page 92), `read_atom/2`, `read_integer/2`, `read_number/2` (section 7.14.2, page 93) and `read_token/2` (section 7.14.3, page 94) to replace any occurrence of a character `InChar` by `OutChar`. However the conversion mechanism should have been previously activated by switching on the `char_conversion` Prolog flag (section 7.22.1, page 133). When `InChar` and `OutChar` are the same, the effect is to remove any conversion of a character `InChar`.

Note that the single character read predicates (e.g. `get_char/2`) never do character conversion. If such behavior is required, it must be explicitly done using `current_char_conversion/2` (section 7.14.13, page 103).

Errors

`InChar` is a variable	`instantiation_error`
`OutChar` is a variable	`instantiation_error`
`InChar` is neither a variable nor a character	`type_error(character, InChar)`
`OutChar` is neither a variable nor a character	`type_error(character, OutChar)`

Portability

ISO predicate. The `type_error(character,...)` is a GNU Prolog behavior, the ISO reference instead defines a `representation_error(character)` in this case. This seems to be an error of the ISO reference since, for many other built-in predicates accepting a character (e.g. `char_code/2`, `put_char/2`), a `type_error` is raised.

The ISO reference implies that if a program calls `current_char_conversion/2`, then modifies the character mapping by calling `char_conversion/2`, and backtracks into the call to `current_char_conversion/2` then the changes are guaranteed not to affect that `current_char_conversion/2` goal. This is not guaranteed by GNU Prolog.

7.14.13 `current_char_conversion/2`

Templates

```
current_char_conversion(?character, ?character)
```

Description

`current_char_conversion(InChar, OutChar)` succeeds if the conversion of `InChar` is `OutChar` according to the character-conversion mapping. In that case, `InChar` and `OutChar` are different. This predicate is re-executable on backtracking.

Errors

`InChar` is neither a variable nor a character	`type_error(character, InChar)`
`OutChar` is neither a variable nor a character	`type_error(character, OutChar)`

Portability

ISO predicate. Same remark as for char_conversion/2 (section 7.14.12, page 102).

7.15 Input/output from/to constant terms

These built-in predicates enable a Prolog term to be input from or output to a Prolog constant term (atom, character list or character code list). All these predicates can be defined using constant term streams (section 7.11, page 82). They are however simpler to use.

7.15.1 `read_term_from_atom/3`, `read_from_atom/2`, `read_token_from_atom/2`

Templates

```
read_term_from_atom(+atom ?term, +read_option_list)
read_from_atom(+atom, ?term)
read_token_from_atom(+atom, ?nonvar)
```

Description

Like `read_term/3`, `read/2` (section 7.14.1, page 92) and `read_token/2` (section 7.14.3, page 94) except that characters are not read from a text-stream but from `Atom`; the atom given as first argument.

Errors

`Atom` is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
see associated predicate errors	(section 7.14.1, page 92) and (section 7.14.3, page 94)

Portability

GNU Prolog predicates.

7.15.2 `read_term_from_chars/3`, `read_from_chars/2`, `read_token_from_chars/2`

Templates

```
read_term_from_chars(+character_list ?term, +read_option_list)
read_from_chars(+character_list, ?term)
read_token_from_chars(+character_list, ?nonvar)
```

Description

Like `read_term/3`, `read/2` (section 7.14.1, page 92) and `read_token/2` (section 7.14.3, page 94) except that characters are not read from a text-stream but from `Chars`; the character list given as first argument.

Errors

`Chars` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Chars` is neither a partial list nor a list	`type_error(list, Chars)`
an element `E` of the `Chars` list is neither a variable nor a character	`type_error(character, E)`
see associated predicate errors	(section 7.14.1, page 92) and (section 7.14.3, page 94)

Portability

GNU Prolog predicates.

7.15.3 `read_term_from_codes/3`, `read_from_codes/2`, `read_token_from_codes/2`

Templates

```
read_term_from_codes(+character_code_list ?term, +read_option_list)
read_from_codes(+character_code_list, ?term)
read_token_from_codes(+character_code_list, ?nonvar)
```

Description

Like `read_term/3`, `read/2` (section 7.14.1, page 92) and `read_token/2` (section 7.14.3, page 94) except that characters are not read from a text-stream but from `Codes`; the character code list given as first argument.

Errors

`Codes` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Codes` is neither a partial list nor a list	`type_error(list, Codes)`
an element `E` of the `Codes` list is neither a variable nor an integer	`type_error(integer, E)`
an element `E` of the `Codes` list is an integer but not a character code	`representation_error(character_code, E)`
see associated predicate errors	(section 7.14.1, page 92) and (section 7.14.3, page 94)

Portability

GNU Prolog predicates.

7.15.4 `write_term_to_atom/3`, `write_to_atom/2`, `writeq_to_atom/2`, `write_canonical_to_atom/2`, `display_to_atom/2`, `print_to_atom/2`, `format_to_atom/3`

Templates

```
write_term_to_atom(?atom, ?term, +write_option_list)
write_to_atom(?atom, ?term)
writeq_to_atom(?atom, ?term)
write_canonical_to_atom(?atom, ?term)
display_to_atom(?atom, ?term)
print_to_atom(?atom, ?term)
format_to_atom(?atom, +character_code_list_or_atom, +list)
```

Description

Similar to `write_term/3`, `write/2`, `writeq/2`, `write_canonical/2`, `display/2`, `print/2` (section 7.14.6, page 96) and `format/3` (section 7.14.7, page 98) except that characters are not written onto a text-stream but are collected as an atom which is then unified with the first argument `Atom`.

Errors

`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
see associated predicate errors	(section 7.14.6, page 96) and (section 7.14.7, page 98)

Portability

GNU Prolog predicates.

7.15.5 `write_term_to_chars/3`, `write_to_chars/2`, `writeq_to_chars/2`, `write_canonical_to_chars/2`, `display_to_chars/2`, `print_to_chars/2`, `format_to_chars/3`

Templates

```
write_term_to_chars(?character_list, ?term, +write_option_list)
write_to_chars(?character_list, ?term)
writeq_to_chars(?character_list, ?term)
write_canonical_to_chars(?character_list, ?term)
display_to_chars(?character_list, ?term)
print_to_chars(?character_list, ?term)
format_to_chars(?character_list, +character_code_list_or_atom, +list)
```

Description

Similar to `write_term/3`, `write/2`, `writeq/2`, `write_canonical/2`, `display/2`, `print/2` (section 7.14.6, page 96) and `format/3` (section 7.14.7, page 98) except that characters are not written onto a text-stream but are collected as a character list which is then unified with the first argument `Chars`.

Errors

`Chars` is neither a partial list nor a list	`type_error(list, Chars)`
see associated predicate errors	(section 7.14.6, page 96) and (section 7.14.7, page 98)

Portability

GNU Prolog predicates.

7.15.6 `write_term_to_codes/3`, `write_to_codes/2`, `writeq_to_codes/2`, `write_canonical_to_codes/2`, `display_to_codes/2`, `print_to_codes/2`, `format_to_codes/3`

Templates

```
write_term_to_codes(?character_code_list, ?term, +write_option_list)
write_to_codes(?character_code_list, ?term)
writeq_to_codes(?character_code_list, ?term)
write_canonical_to_codes(?character_code_list, ?term)
display_to_codes(?character_code_list, ?term)
print_to_codes(?character_code_list, ?term)
format_to_codes(?character_code_list, +character_code_list_or_atom, +list)
```

Description

Similar to `write_term/3`, `write/2`, `writeq/2`, `write_canonical/2`, `display/2`, `print/2` (section 7.14.6, page 96) and `format/3` (section 7.14.7, page 98) except that characters are not written onto a text-stream but are collected as a character code list which is then unified with the first argument `Codes`.

Errors

`Codes` is neither a partial list nor a list	`type_error(list, Codes)`
see associated predicate errors	(section 7.14.6, page 96) and (section 7.14.7, page 98)

Portability

GNU Prolog predicates.

7.16 DEC-10 compatibility input/output

7.16.1 Introduction

The DEC-10 Prolog I/O predicates manipulate streams implicitly since they only refer to current input/output streams (section 7.10.1, page 67). The current input and output streams are initially set to `user_input` and `user_output` respectively. The predicate `see/1` (resp. `tell/1`, `append/1`) can be used for setting the current input (resp. output) stream to newly opened streams for particular files. The predicate `seen/0` (resp. `told/0`) close the current input (resp. output) stream, and resets it to the standard input (resp. output). The predicate `seeing/1` (resp. `telling/1`) is used for retrieving the file name associated with the current input (resp. output) stream. The file name `user` stands for the standard input or output, depending on context (`user_input` and `user_output` can also be used). The DEC-10 Prolog I/O predicates are only provided for compatibility, they are now obsolete and their use is discouraged. The predicates for explicit stream manipulation should be used instead (section 7.10, page 67).

7.16.2 `see/1`, `tell/1`, `append/1`

Templates

```
see(+source_sink)
see(+stream)
tell(+source_sink)
tell(+stream)
append(+source_sink)
append(+stream)
```

Description

`see(FileName)` sets the current input stream to `FileName`. If there is a stream opened by `see/1` associated with the same `FileName` already, then it becomes the current input stream. Otherwise, `FileName` is opened for reading and becomes the current input stream.

`tell(FileName)` sets the current output stream to `FileName`. If there is a stream opened by `tell/1` associated with the same `FileName` already, then it becomes the current output stream. Otherwise, `FileName` is opened for writing and becomes the current output stream.

`append(FileName)` like `tell/1` but `FileName` is opened for writing + append.

A stream-term (obtained with any other built-in predicate) can also be provided as `FileName` to these predicates.

Errors

See errors associated to `open/4` (section 7.10.6, page 70).

Portability

GNU Prolog predicates.

7.16.3 `seeing/1`, `telling/1`

Templates

```
seeing(?source_sink)
telling(?source_sink)
```

Description

`seeing(FileName)` succeeds if `FileName` unifies with the name of the current input file, if it was opened by `see/1`; else with the current input stream-term, if this is not `user_input`, otherwise with `user`.

`telling(FileName)` succeeds if `FileName` unifies with the name of the current output file, if it was opened by `tell/1` or `append/1`; else with the current output stream-term, if this is not `user_output`, otherwise with `user`.

Errors

None.

Portability

GNU Prolog predicates.

7.16.4 `seen/0`, `told/0`

Templates

```
seen
told
```

Description

`seen` closes the current input, and resets it to `user_input`.

`told` closes the current output, and resets it to `user_output`.

Errors

None.

Portability

GNU Prolog predicates.

7.16.5 `get0/1`, `get/1`, `skip/1`

Templates

```
get0(?in_character_code)
get(?in_character_code)
skip(+character_code)
```

Description

`get0(Code)` succeeds if `Code` unifies with the next character code read from the current input stream. Thus it is equivalent to `get_code(Code)` (section 7.12.1, page 85).

`get(Code)` succeeds if `Code` unifies with the next character code read from the current input stream that is not a layout character.

`skip(Code)` skips just past the next character code `Code` from the current input stream.

Errors

See errors for `get_code/2` (section 7.12.1, page 85).

Portability

GNU Prolog predicates.

7.16.6 put/1, tab/1

Templates

```
put(+character_code)
tab(+evaluable)
```

Description

`put(Code)` writes the character whose code is `Code` onto the current output stream. It is equivalent to `put_code(Code)` (section 7.12.5, page 88).

`tab(N)` writes `N` spaces onto the current output stream. `N` may be an arithmetic expression.

Errors

See errors for `put_code/2` (section 7.12.5, page 88) and for arithmetic expressions (section 7.6.1, page 57).

Portability

GNU Prolog predicates.

7.17 Term expansion

7.17.1 Definite clause grammars

Definite clause grammars are a useful notation to express grammar rules. However the ISO reference does not include them, so they should be considered as a system dependent feature. Definite clause grammars are an extension of context-free grammars. A grammar rule is of the form:

head `-->` body.

`-->` is a predefined infix operator (section 7.14.10, page 100).

Here are some features of definite clause grammars:

- a non-terminal symbol may be any callable term.
- a terminal symbol may be any Prolog term and is written as a list. The empty list represents an empty sequence of terminals.
- a sequence is expressed using the Prolog conjunction operator ((',')/2).
- the head of a grammar rule consists of a non-terminal optionally followed by a sequence of terminals (i.e. a Prolog list).
- the body of a grammar rule consists of a sequence of non-terminals, terminals, predicate call, disjunction (using `;/2`), if-then (using `(->)/2`) or cut (using `!`).
- a predicate call must be enclosed in curly brackets (using `{}/1`). This makes it possible to express an extra condition.

A grammar rule is nothing but a "syntactic sugar" for a Prolog clause. Each grammar rule accepts as input a list of terminals (tokens), parses a prefix of this list and gives as output the rest of this list (possibly enlarged). This rest is generally parsed later. So, each a grammar rule is translated into a Prolog clause that explicitly the manages the list. Two arguments are then added: the input list (`Start`) and the output list (`Stop`). For instance:

```
p --> q.
```

is translated into:

```
p(Start, End) :- q(Start, End).
```

Extra arguments can be provided and the body of the rule can contain several non-terminals. Example:

```
p(X, Y) -->
        q(X),
        r(X, Y),
        s(Y).
```

is translated into:

```
p(X, Y, Start, End) :-
        q(X, Start, A),
        r(X, Y, A, B),
        s(Y, B, End).
```

Terminals are translated using unification:

```
assign(X,Y) --> left(X), [:=], right(Y), [;].
```

is translated into:

```
assign(X,Y,Start,End) :-
        left(X, Start, A),
        A=[:=|B],
        right(Y, B, C),
        C=[;|End].
```

Terminals appearing on the left-hand side of a rule are connected to the output argument of the head.

It is possible to include a call to a prolog predicate enclosing it in curly brackets (to distinguish them from non-terminals):

```
assign(X,Y) --> left(X), [:=], right(Y0), {Y is Y0 }, [;].
```

is translated into:

```
assign(X,Y,Start,End) :-
        left(X, Start, A),
        A=[:=|B],
        right(Y0, B, C),
        Y is Y0,
        C=[;|End].
```

Cut, disjunction and if-then(-else) are translated literally (and do not need to be enclosed in curly brackets).

7.17.2 `expand_term/2`, `term_expansion/2`

Templates

```
expand_term(?term, ?term)
term_expansion(?term, ?term)
```

Description

`expand_term(Term1, Term2)` succeeds if `Term2` is a transformation of `Term1`. The transformation steps are as follows:

- if `Term1` is a variable, it is unified with `Term2`
- if `term_expansion(Term1, Term2)` succeeds `Term2` is assumed to be the transformation of `Term1`.
- if `Term1` is a DCG then `Term2` is its translation (section 7.17.1, page 109).
- otherwise `Term2` is unified with `Term1`.

`term_expansion(Term1, Term2)` is a hook predicate allowing the user to define a specific transformation.

The GNU Prolog compiler (section 3.4, page 20) automatically calls `expand_term/2` on each `Term1` read in. However, in the current release, only DCG transformation are done by the compiler (i.e. `term_expansion/2` cannot be used). To use `term_expansion/2`, it is necessary to call `expand_term/2` explicitly.

Errors

None.

Portability

GNU Prolog predicate.

7.17.3 `phrase/3`, `phrase/2`

Templates

```
phrase(?term, ?list, ?list)
phrase(?term, ?list)
```

Description

`phrase(Phrase, List, Remainder)` succeeds if the list `List` is in the language defined by the grammar rule body `Phrase`. `Remainder` is what remains of the list after a phrase has been found.

`phrase(Phrase, List)` is equivalent to `phrase(Phrase, List, [])`.

Errors

`List` is neither a list nor a partial list	`type_error(list, List)`
`Remainder` is neither a list nor a partial list	`type_error(list, Remainder)`

Portability

GNU Prolog predicates.

7.18 Logic, control and exceptions

7.18.1 abort/0, stop/0, top_level/0, break/0, halt/1, halt/0

Templates

```
abort
stop
top_level
break
halt(+integer)
halt
```

Description

`abort` aborts the current execution. If this execution was initiated under a top-level the control is given back to the top-level and the message `{execution aborted}` is displayed. Otherwise, e.g. execution started by a `initialization/1` directive (section 6.1.13, page 45), `abort/0` is equivalent to `halt(1)` (see below).

`stop` stops the current execution. If this execution was initiated under a top-level the control is given back to the top-level. Otherwise, `stop/0` is equivalent to `halt(0)` (see below).

`top_level` starts a new recursive top-level (including the banner display). To end this new top-level simply type the end-of-file key sequence (`Ctl-D`) or its term representation: `end_of_file.`

`break` invokes a recursive top-level (no banner is displayed). To end this new level simply type the end-of-file key sequence (`Ctl-D`) or its term representation: `end_of_file.`

`halt(Status)` causes the GNU Prolog process to immediately exit back to the shell with the return code `Status`.

`halt` is equivalent to `halt(0)`.

Errors

`Status` is a variable	`instantiation_error`
`Status` is neither a variable nor an integer	`type_error(integer, Status)`

Portability

`halt/1` and `halt/0` are ISO predicates. `abort/0`, `stop/0`, `top_level/0` and `break/0` are GNU Prolog predicates.

7.18.2 `once/1`, `(\+)/1` - not provable, `call_with_args/1-11`, `call/2`

Templates

```
once(+callable_term)
\+(+callable_term)
call_with_args(+atom, +term,..., +term)
call(+callable_term, ?boolean)
```

Description

`once(Goal)` succeeds if `call(Goal)` succeeds. However `once/1` is not re-executable on backtracking since all alternatives of `Goal` are cut. `once(Goal)` is equivalent to `call(Goal), !`.

`\+ Goal` succeeds if `call(Goal)` fails and fails otherwise. This built-in predicate gives negation by failure.

`call_with_args(Functor, Arg1,..., ArgN)` calls the goal whose functor is `Functor` and whose arguments are `Arg1`,..., `ArgN` ($0 \leq$ `N` ≤ 10).

`call(Goal, Deterministic)` succeeds if `call(Goal)` succeeds and unifies `Deterministic` with `true` if `Goal` has not created any choice-points, with `false` otherwise.

`\+` is a predefined prefix operator (section 7.14.10, page 100).

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`
`Functor` is a variable	`instantiation_error`
`Functor` is neither a variable nor an atom	`type_error(atom, Functor)`
`Deterministic` is neither a variable nor a boolean	`type_error(boolean, Deterministic)`

Portability

`once/1` and `(\+)/1` are ISO predicates, `call_with_args/1-11` and `call/2` are GNU Prolog predicates.

7.18.3 `repeat/0`

Templates

```
repeat
```

Description

`repeat` generates an infinite sequence of backtracking choices. The purpose is to repeatedly perform some action on elements which are somehow generated, e.g. by reading them from a stream, until some test becomes true. Repeat loops cannot contribute to the logic of the program. They are only meaningful if the action involves side-effects. The only reason for using repeat loops instead of a more natural tail-recursive formulation is efficiency: when the test fails back, the Prolog engine immediately reclaims any working storage consumed since the call to `repeat/0`.

Errors

None.

Portability

ISO predicate.

7.18.4 `for/3`

Templates

    for(?integer, +integer, +integer)

Description

`for(Counter, Lower, Upper)` generates an sequence of backtracking choices instantiating `Counter` to the values `Lower`, `Lower+1`,..., `Upper`.

Errors

`Counter` is neither a variable nor an integer	`type_error(integer, Counter)`
`Lower` is a variable	`instantiation_error`
`Lower` is neither a variable nor an integer	`type_error(integer, Lower)`
`Upper` is a variable	`instantiation_error`
`Upper` is neither a variable nor an integer	`type_error(integer, Upper)`

Portability

GNU Prolog predicate.

7.19 Atomic term processing

These built-in predicates enable atomic terms to be processed as a sequence of characters and character codes. Facilities exist to split and join atoms, to convert a single character to and from the corresponding character code, and to convert a number to and from a list of characters and character codes.

7.19.1 `atom_length/2`

Templates

    atom_length(+atom, ?integer)

Description

`atom_length(Atom, Length)` succeeds if `Length` unifies with the number of characters of the name of `Atom`.

Errors

`Atom` is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Length` is neither a variable nor an integer	`type_error(integer, Length)`
`Length` is an integer < 0	`domain_error(not_less_than_zero, Length)`

Portability

ISO predicate.

7.19.2 atom_concat/3

Templates

```
atom_concat(+atom, +atom, ?atom)
atom_concat(?atom, ?atom, +atom)
```

Description

`atom_concat(Atom1, Atom2, Atom12)` succeeds if the name of `Atom12` is the concatenation of the name of `Atom1` with the name of `Atom1`. This predicate is re-executable on backtracking (e.g. if `Atom12` is instantiated and both `Atom1` and `Atom2` are variables).

Errors

`Atom1` and `Atom12` are variables	`instantiation_error`
`Atom2` and `Atom12` are variables	`instantiation_error`
`Atom1` is neither a variable nor an atom	`type_error(atom, Atom1)`
`Atom2` is neither a variable nor an atom	`type_error(atom, Atom2)`
`Atom12` is neither a variable nor an atom	`type_error(atom, Atom12)`

Portability

ISO predicate.

7.19.3 sub_atom/5

Templates

```
sub_atom(+atom, ?integer, ?integer, ?integer, ?atom)
```

Description

`sub_atom(Atom, Before, Length, After, SubAtom)` succeeds if atom `Atom` can be split into three atoms, `AtomL`, `SubAtom` and `AtomR` such that `Before` is the number of characters of the name of `AtomL`, `Length` is the number of characters of the name of `SubAtom` and `After` is the number of characters of the name of `AtomR`. This predicate is re-executable on backtracking.

Errors

`Atom` is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`SubAtom` is neither a variable nor an atom	`type_error(atom, SubAtom)`
`Before` is neither a variable nor an integer	`type_error(integer, Before)`
`Length` is neither a variable nor an integer	`type_error(integer, Length)`
`After` is neither a variable nor an integer	`type_error(integer, After)`
`Before` is an integer < 0	`domain_error(not_less_than_zero, Before)`
`Length` is an integer < 0	`domain_error(not_less_than_zero, Length)`
`After` is an integer < 0	`domain_error(not_less_than_zero, After)`

Portability

ISO predicate.

7.19.4 char_code/2

Templates

```
char_code(+character, ?character_code)
char_code(-character, +character_code)
```

Description

`char_code(Char, Code)` succeeds if the character code for the one-char atom `Char` is `Code`.

Errors

`Char` and `Code` are variables	`instantiation_error`
`Char` is neither a variable nor a one-char atom	`type_error(character, Char)`
`Code` is neither a variable nor an integer	`type_error(integer, Code)`
`Code` is an integer but not a character code	`representation_error(character_code)`

Portability

ISO predicate.

7.19.5 lower_upper/2

Templates

```
lower_upper(+character, ?character)
lower_upper(-character, +character)
```

Description

`lower_upper(Char1, Char2)` succeeds if `Char1` and `Char2` are one-char atoms and if `Char2` is the upper conversion of `Char1`. If `Char1` (resp. `Char2`) is a character that is not a lower (resp. upper) letter then `Char2` is equal to `Char1`.

Errors

`Char1` and `Char2` are variables	`instantiation_error`
`Char1` is neither a variable nor a one-char atom	`type_error(character, Char1)`
`Char2` is neither a variable nor a one-char atom	`type_error(character, Char2)`

Portability

GNU Prolog predicate.

7.19.6 atom_chars/2, atom_codes/2

Templates

```
atom_chars(+atom, ?character_list)
atom_chars(-atom, +character_list)
atom_codes(+atom, ?character_code_list)
atom_codes(-atom, +character_code_list)
```

Description

`atom_chars(Atom, Chars)` succeeds if `Chars` is the list of one-char atoms whose names are the successive characters of the name of `Atom`.

`atom_codes(Atom, Codes)` is similar to `atom_chars/2` but deals with a list of character codes.

Errors

`Atom` is a variable and `Chars` (or `Codes`) is a partial list or a list with an element which is a variable	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Chars` is neither a list nor a partial list	`type_error(list, Chars)`
`Codes` is neither a list nor a partial list	`type_error(list, Codes)`
`Atom` is a variable and an element `E` of the list `Chars` is neither a variable nor a one-char atom	`type_error(character, E)`
`Atom` is a variable and an element `E` of the list `Codes` is neither a variable nor an integer	`type_error(integer, E)`
`Atom` is a variable and an element `E` of the list `Codes` is an integer but not a character code	`representation_error(character_code)`

Portability

ISO predicates. The ISO reference only causes a `type_error(list, Chars)` if `Atom` is a variable and `Chars` is neither a list nor a partial list. GNU Prolog always checks if `Chars` is a list. Similarly for `Codes`. The `type_error(integer, E)` when an element `E` of the `Codes` is not an integer is a GNU Prolog extension. This seems to be an omission in the ISO reference since this error is detected for many other built-in predicates accepting a character code (e.g. `char_code/2`, `put_code/2`).

7.19.7 `number_atom/2`, `number_chars/2`, `number_codes/2`

Templates

```
number_atom(+number, ?atom)
number_atom(-number, +atom)
number_chars(+number, ?character_list)
number_chars(-number, +character_list)
number_codes(+number, ?character_code_list)
number_codes(-number, +character_code_list)
```

Description

`number_atom(Number, Atom)` succeeds if `Atom` is an atom whose name corresponds to the characters of `Number`.

`number_chars(Number, Chars)` is similar to `number_atom/2` but deals with a list of character codes.

`number_codes(Number, Codes)` is similar to `number_atom/2` but deals with a list of characters.

Errors

`Number` and `Atom` are variables	`instantiation_error`
`Number` is a variable and `Chars` (or `Codes`) is a partial list or a list with an element which is a variable	`instantiation_error`
`Number` is neither a variable nor an number	`type_error(number, Number)`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Number` is a variable and `Chars` is neither a list nor a partial list	`type_error(list, Chars)`
`Number` is a variable and `Codes` is neither a list nor a partial list	`type_error(list, Codes)`
`Number` is a variable and an element `E` of the list `Chars` is neither a variable nor a one-char atom	`type_error(character, E)`
`Number` is a variable and an element `E` of the list `Codes` is neither a variable nor an integer	`type_error(integer, E)`
`Number` is a variable and an element `E` of the list `Codes` is an integer but not a character code	`representation_error(character_code)`
`Number` is a variable, `Atom` (or `Chars` or `Codes`) cannot be parsed as a number and the value of the `syntax_error` Prolog flag is `error` (section 7.22.1, page 133)	`syntax_error(`*atom explaining the error*`)`

Portability

`number_atom/2` is a GNU Prolog predicate. `number_chars/2` and `number_codes/2` are ISO predicates.

GNU Prolog only raises an error about an element `E` of the `Chars` (or `Codes`) list when `Number` is a variable while the ISO reference always check this. This seems an error since the list itself is only checked if `Number` is a variable.

The `type_error(integer, E)` when an element `E` of the `Codes` is not an integer is a GNU Prolog extension. This seems to be an omission in the ISO reference since this error is detected for many other built-in predicates accepting a character code (e.g. `char_code/2`, `put_code/2`).

7.19.8 name/2

Templates

```
name(+atomic, ?character_code_list)
name(-atomic, +character_code_list)
```

Description

`name(Constant, Codes)` succeeds if `Codes` is a list whose elements are the character codes corresponding to the successive characters of `Constant` (a number or an atom). However, there atoms are for which `name(Constant, Codes)` is true, but which will not be constructed if `name/2` is called with `Constant` uninstantiated, e.g. the atom `'1024'`. For this reason the use of `name/2` is discouraged and should be limited to compatibility purposes. It is preferable to use atom_codes/2 (section 7.19.6, page 116) or number_chars/2 (section 7.19.7, page 117).

Errors

`Constant` is a variable and `Codes` is a partial list or a list with an element which is a variable	`instantiation_error`
`Constant` is neither a variable nor an atomic term	`type_error(atomic, Constant)`
`Constant` is a variable and `Codes` is neither a list nor a partial list	`type_error(list, Codes)`
`Constant` is a variable and an element `E` of the list `Codes` is neither a variable nor an integer	`type_error(integer, E)`
`Constant` is a variable and an element `E` of the list `Codes` is an integer but not a character code	`representation_error(character_code)`

Portability

GNU Prolog predicate.

7.19.9 `atom_hash/2`

Templates

```
atom_hash(+atom, ?integer)
atom_hash(?atom, +integer)
```

Description

`atom_hash(Atom, Hash)` succeeds if `Hash` is the internal key associated to `Atom` (an existing atom). The internal key of an atom is a unique integer ≥ 0 and $<$ to the `max_atom` Prolog flag (section 7.22.1, page 133).

Errors

`Atom` and `Hash` are both variables	`instantiation_error`
`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Hash` is neither a variable nor an integer	`type_error(integer, Hash)`
`Hash` is an integer < 0	`domain_error(not_less_than_zero, Hash)`

Portability

GNU Prolog predicate.

7.19.10 `new_atom/3`, `new_atom/2`, `new_atom/1`

Templates

```
new_atom(+atom, +integer, -atom)
new_atom(+atom, -atom)
new_atom(-atom)
```

Description

`new_atom(Prefix, Hash, Atom)` unifies `Atom` with a new atom whose name begins with the characters of the name of `Prefix` and whose internal key is `Hash` (section 7.19.9, page 119). This predicate is then a symbol generator. It is guaranteed that `Atom` does not exist before the invocation of `new_atom/3`. The characters appended to `Prefix` to form `Atom` are in: `A-Z` (capital letter), `a-z` (small letter), `0-9` (digit), `#`, `$`, `&`, `_`, `@`.

`new_atom/2` is similar to `new_atom/3`, but the atom generated can have any (free) internal key.

`new_atom/1` is similar to `new_atom(atom_, Atom)`, i.e. the generated atom begins with `atom_`.

Errors

`Prefix` is a variable	`instantiation_error`
`Hash` is a variable	`instantiation_error`
`Prefix` is neither a variable nor an atom	`type_error(atom, Prefix)`
`Hash` is neither a variable nor an integer	`type_error(integer, Hash)`
`Hash` is an integer < 0	`domain_error(not_less_than_zero, Hash)`
`Atom` is not a variable	`type_error(variable, Atom)`

Portability

GNU Prolog predicate.

7.19.11 `current_atom/1`

Templates

```
current_atom(?atom)
```

Description

`current_atom(Atom)` succeeds if there exists an atom that unifies with `Atom`. All atoms are found except those beginning with a `'$'` (system atoms). This predicate is re-executable on backtracking.

Errors

`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`

Portability

GNU Prolog predicate.

7.19.12 `atom_property/2`

Templates

```
atom_property(?atom, ?atom_property)
```

Description

`atom_property(Atom, Property)` succeeds if `current_atom(Atom)` succeeds (section 7.19.11, page 120) and if `Property` unifies with one of the properties of the atom. This predicate is re-executable on backtracking.

Atom properties:

- `length(Length)`: `Length` is the length of the name of the atom.
- `hash(Hash)`: `Hash` is the internal key of the atom (section 7.19.9, page 119).
- `prefix_op`: if there is a prefix operator currently defined with this name.
- `infix_op`: if there is an infix operator currently defined with this name.
- `postfix_op`: if there is a postfix operator currently defined with this name.

- `needs_quotes`: if the atom must be quoted to be read later.
- `needs_scan`: if the atom must be scanned when output to be read later (e.g. contains special characters that must be output with a \ escape sequence).

Errors

`Atom` is neither a variable nor an atom	`type_error(atom, Atom)`
`Property` is neither a variable nor a n atom property term	`domain_error(atom_property, Property)`
`Property` = `length(E)` or `hash(E)` and `E` is neither a variable nor an integer	`type_error(integer, E)`

Portability

GNU Prolog predicate.

7.20 List processing

These predicates manipulate lists. They are bootstrapped predicates (i.e. written in Prolog) and no error cases are tested (for the moment). However, since they are written in Prolog using other built-in predicates, some errors can occur due to those built-in predicates.

7.20.1 append/3

Templates

```
append(?list, ?list, ?list)
```

Description

`append(List1, List2, List12)` succeeds if the concatenation of the list `List1` and the list `List2` is the list `List12`. This predicate is re-executable on backtracking (e.g. if `List12` is instantiated and both `List1` and `List2` are variable).

Errors

None.

Portability

GNU Prolog predicate.

7.20.2 member/2, memberchk/2

Templates

```
member(?term, ?list)
memberchk(?term, ?list)
```

Description

`member(Element, List)` succeeds if `Element` belongs to the `List`. This predicate is re-executable on backtracking and can be thus used to enumerate the elements of `List`.

`memberchk/2` is similar to `member/2` but only succeeds once.

Errors

None.

Portability

GNU Prolog predicate.

7.20.3 `reverse/2`

Templates

```
reverse(?list, ?list)
```

Description

`reverse(List1, List2)` succeeds if `List2` unifies with the list `List1` in reverse order.

Errors

None.

Portability

GNU Prolog predicate.

7.20.4 `delete/3`, `select/3`

Templates

```
delete(?list, ?term, ?list)
select(?term, ?list, ?list)
```

Description

`delete(List1, Element, List2)` removes all occurrences of `Element` in `List1` to provide `List2`. A strict term equality is required, cf. `(==)/2` (section 7.3.2, page 51).

`select(Element, List1, List2)` removes one occurrence of `Element` in `List1` to provide `List2`. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

7.20.5 `permutation/2`

Templates

```
permutation(?list, ?list)
```

Description

`permutation(List1, List2)` succeeds if `List2` is a permutation of the elements of `List1`. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

7.20.6 `prefix/2`, `suffix/2`

Templates

```
prefix(?list, ?list)
suffix(?list, ?list)
```

Description

`prefix(Prefix, List)` succeeds if `Prefix` is a prefix of `List`. This predicate is re-executable on backtracking.

`suffix(Suffix, List)` succeeds if `Suffix` is a suffix of `List`. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

7.20.7 `sublist/2`

Templates

```
sublist(?list, ?list)
```

Description

`sublist(List1, List2)` succeeds if `List2` is a sub-list of `List1`. This predicate is re-executable on backtracking.

Errors

None.

Portability

GNU Prolog predicate.

7.20.8 `last/2`

Templates

```
last(?list, ?term)
```

Description

`last(List, Element)` succeeds if `Element` is the last element of `List`.

Errors

None.

Portability

GNU Prolog predicate.

7.20.9 `length/2`

Templates

```
length(?list, ?integer)
```

Description

`length(List, Length)` succeeds if `Length` is the length of `List`.

Errors

None.

Portability

GNU Prolog predicate.

7.20.10 `nth/3`

Templates

```
nth(?integer, ?list, ?term)
```

Description

`nth(N, List, Element)` succeeds if the `N`*th* argument of `List` is `Element`.

Errors

None.

Portability

GNU Prolog predicate.

7.20.11 `max_list/2`, `min_list/2`, `sum_list/2`

Templates

```
min_list(+list, ?number)
max_list(+list, ?number)
sum_list(+list, ?number)
```

Description

`min_list(List, Min)` succeeds if `Min` is the smallest number in `List`.

`max_list(List, Max)` succeeds if `Max` is the largest number in `List`.

`sum_list(List, Sum)` succeeds if `Sum` is the sum of all the elements in `List`.

`List` must be a list of arithmetic evaluable terms (section 7.6.1, page 57).

Errors

None.

Portability

GNU Prolog predicate.

7.20.12 `sort/2`, `sort0/2`, `keysort/2` `sort/1`, `sort0/1`, `keysort/1`

Templates

```
sort(+list, ?list)
sort0(+list, ?list)
keysort(+list, ?list)
sort(+list)
sort0(+list)
keysort(+list)
```

Description

`sort(List1, List2)` succeeds if `List2` is the sorted list corresponding to `List1` where duplicate elements are merged.

`sort0/2` is similar to `sort/2` except that duplicate elements are not merged.

`keysort(List1, List2)` succeeds if `List2` is the sorted list of `List1` according to the keys. The list `List1` consists of items of the form `Key-Value`. These items are sorted according to the value of `Key` yielding the `List2`. Duplicate keys are not merged. This predicate is stable, i.e. if `K-A` occurs before `K-B` in the input, then `K-A` will occur before `K-B` in the output.

`sort/1`, `sort0/1` and `keysort/1` are similar to `sort/2`, `sort0/2` and `keysort/2` but achieve a sort in-place destructing the original `List1` (this in-place assignment is not undone at backtracking). The sorted list occupies the same memory space as the original list (saving thus memory consumption).

The time complexity of these sorts is $O(N\ log\ N)$, N being the length of the list to sort.

These predicates refer to the standard ordering of terms (section 7.3.1, page 51).

Errors

`List1` is a partial list	`instantiation_error`
`List1` is neither a partial list nor a list	`type_error(list, List1)`
`List2` is neither a partial list nor a list	`type_error(list, List2)`

Portability

GNU Prolog predicates.

7.21 Global variables

7.21.1 Introduction

GNU Prolog provides a simple and powerful way to assign and read global variables. A global variable is associated to each atom, its initial value is the integer 0. A global variable can store 3 kinds of objects:

- a copy of a term (the assignment can be made backtrackable or not).
- a link to a term (the assignment is always backtrackable).
- an array of objects (recursively).

The space necessary for copies and arrays is dynamically allocated and recovered as soon as possible. For instance, when an atom is associated to a global variable whose current value is an array, the space for this array is recovered (unless the assignment is to be undone when backtracking occurs).

When a link to a term is associated to a global variable, the reference to this term is stored and thus the original term is returned when the content of the variable is read.

Global variable naming convention: a global variable is referenced by an atom.

If the variable contains an array, an index (ranging from 0) can be provided using a compound term whose principal functor is the correponding atom and the argument is the index. In case of a multi-dimensional array, each index is given as the arguments of the compound term.

If the variable contains a term (link or copy), it is possible to only reference a sub-term by giving its argument number (also called argument selector). Such a sub-term is specified using a compound term whose principal functor is `-/2` and whose first argument is a global variable name and the second argument is the argument number (from 1). This can be applied recursively to specify a sub-term of any depth. In case of a list, a argument number I represents the Ith element of the list. In the rest of this section we use the operator notation since `-` is a predefined infix operator (section 7.14.10, page 100).

In the following, *`GVarName`* represents a reference to a global variable and its syntax is as follows:

`GVarName`	::=	*`atom`*	whole content of a variable
		`atom(Integer,...,Integer)`	element of an array
		`GVarName-Integer`	sub-term selection
`Integer`	::=	*`integer`*	immediate value
		`GVarName`	indirect value

When a *`GVarName`* is used as an index or an argument number (i.e. indirection), the value of this variable must be an integer.

Here are some examples of the naming convention:

`a`	the content of variable associated to `a` (any kind)
`t(1)`	the 2nd element of the array associated to `t`
`t(k)`	if the value associated to `k` is I, the Ith element of the array associated to `t`
`a-1-2`	if the value associated to `a` is `f(g(a,b,c),2)`, the sub-term `b`

Here are the errors associated to global variable names and common to all predicates.

`GVarName` is a variable	`instantiation_error`
`GVarName` is neither a variable nor a callable term	`type_error(callable, GVarName)`
`GVarName` contains an invalid argument number (or `GVarName` is an array)	`domain_error(g_argument_selector, GVarName)`
`GVarName` contains an invalid index (or `GVarName` is not an array)	`domain_error(g_array_index, GVarName)`
`GVarName` is used as an indirect index or argument selector and is not an integer	`type_error(integer, GVarName)`

Arrays: the predicates `g_assign/2`, `g_assignb/2` and `g_link/2` (section 7.21.2, page 127) can be used to create an array. They recognize some terms as values. For instance, a compound term with principal functor `g_array` is used to define an array of fixed size. There are 3 forms for the term `g_array`:

- `g_array(Size)`: if `Size` is an integer > 0 then defines an array of `Size` elements which are all initialized with the integer `0`.
- `g_array(Size, Initial)`: as above but the elements are initialized with the term `Initial` instead of 0. `Initial` can contain other array definitions allowing thus for multi-dimensional arrays.
- `g_array(List)`: as above if `List` is a list of length `Size` except that the elements of the array are initialized according to the elements of `List` (which can contain other array definitions).

An array can be extended explicitely using a compound term with principal functor `g_array_extend` which accept the same 3 forms detailed above. In that case, the existing elements of the array are not initialized. If `g_array_extend` is used with an object which is not an array it is similar to `g_array`.

Finally, an array can be *automatically* expanded when needed. The programmer does not need to explicitely control the expansion of an automatic array. An array is expanded as soon as an index is outside the current size of this array. Such an array is defined using a compound term with principal functor `g_array_auto`:

- `g_array_auto(Size)`: if `Size` is an integer > 0 then defines an automatic array whose initial size is `Size`. All elements are initialized with the integer `0`. Elements created during implicit expansions will be initialized with `0`.
- `g_array_auto(Size, Initial)`: as above but the elements are initialized with the term `Initial` instead of 0. `Initial` can contain other array definitions allowing thus for multi-dimensional arrays. Elements created during implicit expansions will be initialized with `Initial`.
- `g_array_auto(List)`: as above if `List` is a list of length `Size` except that the elements of the array are initialized according to the elements of `List` (which can contain other array definitions). Elements created during implicit expansions will be initialized with `0`.

In any case, when an array is read, a term of the form `g_array([Elem0,..., ElemSize-1])` is returned.

Some examples using global variables are presented later (section 7.21.7, page 130).

7.21.2 `g_assign/2`, `g_assignb/2`, `g_link/2`

Templates

```
g_assign(+callable_term, ?term)
g_assignb(+callable_term, ?term)
g_link(+callable_term, ?term)
```

Description

`g_assign(GVarName, Value)` assigns a copy of the term `Value` to `GVarName`. This assignment is not undone when backtracking occurs.

`g_assignb/2` is similar to `g_assign/2` but the assignment is undone at backtracking.

`g_link(GVarName, Value)` makes a link between `GVarName` to the term `Value`. This allows the user to give a name to any Prolog term (in particular non-ground terms). Such an assignment is always undone when backtracking occurs (since the term may no longer exist). If `Value` is an atom or an integer, `g_link/2` and `g_assignb/2` have the same behavior. Since `g_link/2` only handles links to existing terms it does not require extra memory space and is not expensive in terms of execution time.

NB: argument selectors can only be used with g_assign/2 (i.e. when using an argument selector inside an assignment, this one must not be backtrackable).

Errors

See common errors detailed in the introduction (section 7.21.1, page 126)

`GVarName` contains an argument selector and the assignment is backtrackable	`domain_error(g_argument_selector, GVarName)`

Portability

GNU Prolog predicates.

7.21.3 `g_read/2`

Templates

```
g_read(+callable_term, ?term)
```

Description

`g_read(GVarName, Value)` unifies `Value` with the term assigned to `GVarName`.

Errors

See common errors detailed in the introduction (section 7.21.1, page 126)

Portability

GNU Prolog predicate.

7.21.4 `g_array_size/2`

Templates

```
g_array_size(+callable_term, ?integer)
```

Description

`g_array_size(GVarName, Value)` unifies `Size` with the dimension (an integer > 0) of the array assigned to `GVarName`. Fails if `GVarName` is not an array.

Errors

See common errors detailed in the introduction (section 7.21.1, page 126)

`Size` is neither a variable nor an integer	`type_error(integer, Size)`

Portability

GNU Prolog predicate.

7.21.5 g_inc/3, g_inc/2, g_inco/2, g_inc/1, g_dec/3, g_dec/2, g_deco/2, g_dec/1

Templates

```
g_inc(+callable_term, ?integer, ?integer)
g_inc(+callable_term, ?integer)
g_inco(+callable_term, ?integer)
g_inc(+callable_term)
g_dec(+callable_term, ?integer, ?integer)
g_dec(+callable_term, ?integer)
g_deco(+callable_term, ?integer)
g_dec(+callable_term)
```

Description

`g_inc(GVarName, Old, New)` unifies `Old` with the integer assigned to `GVarName`, increments `GVarName` and then unifies `New` with the incremented value.

`g_inc(GVarName, New)` is equivalent to `g_inc(GVarName, _, New)`.

`g_inco(GVarName, Old)` is equivalent to `g_inc(GVarName, Old, _)`.

`g_inc(GVarName)` is equivalent to `g_inc(GVarName, _, _)`.

Predicates `g_dec` are similar but decrement the content of `GVarName` instead.

Errors

See common errors detailed in the introduction (section 7.21.1, page 126)

`Old` is neither a variable nor an integer	`type_error(integer, Old)`
`New` is neither a variable nor an integer	`type_error(integer, New)`
`GVarName` stores an array	`type_error(integer, g_array)`
`GVarName` stores a term `T` which is not an integer	`type_error(integer, T)`

Portability

GNU Prolog predicates.

7.21.6 g_set_bit/2, g_reset_bit/2, g_test_set_bit/2, g_test_reset_bit/2

Templates

```
g_set_bit(+callable_term, +integer)
g_reset_bit(+callable_term, +integer)
g_test_set_bit(+callable_term, +integer)
g_test_reset_bit(+callable_term, +integer)
```

Description

`g_set_bit(GVarName, Bit)` sets to 1 the bit number specified by `Bit` of the integer assigned to `GVarName` to 1. Bit numbers range from 0 to the maximum number allowed for integers (this is architecture dependent). If `Bit` is greater than this limit, the modulo with this limit is taken.

`g_reset_bit(GVarName, Bit)` is similar to `g_set_bit/2` but sets the specified bit to 0.

`g_test_set_bit/2` succeeds if the specified bit is set to 1.

`g_test_reset_bit/2` succeeds if the specified bit is set to 0.

Errors

See common errors detailed in the introduction (section 7.21.1, page 126)

`Bit` is a variable	`instantiation_error`
`Bit` is neither a variable nor an integer	`type_error(integer, Bit)`
`Bit` is an integer < 0	`domain_error(not_less_than_zero, Bit)`
`GVarName` stores an array	`type_error(integer, g_array)`
`GVarName` stores a term `T` which is not an integer	`type_error(integer, T)`

Portability

GNU Prolog predicates.

7.21.7 Examples

Simulating `g_inc/3`: this predicate behaves like: global variable:

```
my_g_inc(Var, Old, New) :-
        g_read(Var, Old),
        N is Value + 1,
        g_assign(Var, X),
New = N.
```

The query: `my_g_inc(c, X, _)` will succeed unifying `X` with `0`, another call to `my_g_inc(a, Y, _)` will then unify `Y` with `1`, and so on.

Difference between `g_assign/2` and `g_assignb/2`: `g_assign/2` does not undo its assignment when backtracking occurs whereas `g_assignb/2` undoes it.

```
test(Old) :-                       testb(Old) :-
        g_assign(x,1),                     g_assign(x,1),
        (   g_read(x, Old),                (   g_read(x, Old),
            g_assign(x, 2)                     g_assignb(x, 2)
        ;   g_read(x, Old),                ;   g_read(x, Old),
            g_assign(x, 3)                     g_assign(x, 3)
        ).                                 ).
```

The query `test(Old)` will succeed unifying `Old` with `1` and on backtracking with `2` (i.e. the assignment of the value `2` has not been undone). The query `testb(Old)` will succeed unifying `Old` with `1` and on backtracking with `1` (i.e. the assignment of the value `2` has been undone).

Difference between `g_assign/2` and `g_link/2`: `g_assign/2` (and `g_assignb/2`) creates a copy of the term whereas `g_link/2` does not. `g_link/2` can be used to avoid passing big data structures (e.g. dictionaries,...) as arguments to predicates.

```
test(B) :-                         test(B) :-
        g_assign(b, f(X)),                 g_link(b, f(X)),
        X = 12,                            X = 12,
        g_read(b, B).                      g_read(b, B).
```

The query `test(B)` will succeed unifying `B` with `f(_)` (`g_assign/2` assigns a copy of the value). The query `testl(B)` will succeed unifying `B` with `f(12)` (`g_link/2` assigns a pointer to the term).

Simple array definition: here are some queries to show how arrays can be handled:

```
| ?- g_assign(w, g_array(3)), g_read(w, X).

X = g_array([0,0,0])

| ?- g_assign(w(0), 16), g_assign(w(1), 32), g_assign(w(2), 64), g_read(w, X).

X = g_array([16,32,64])
```

this is equivalent to:

```
| ?- g_assign(k, g_array([16,32,64])), g_read(k, X).

X = g_array([16,32,64])

| ?- g_assign(k, g_array(3,null)), g_read(k, X), g_array_size(k, S).

S = 3
X = g_array([null,null,null])
```

2-D array definition:

```
| ?- g_assign(w, g_array(2, g_array(3))), g_read(w, X).

X = g_array([g_array([0,0,0]),g_array([0,0,0])])

| ?- (   for(I,0,1), for(J,0,2), K is I*3+J, g_assign(w(I,J), K),
         fail
     ;   g_read(w, X)
     ).

X = g_array([g_array([0,1,2]),g_array([3,4,5])])

| ?- g_read(w(1),X).

X = g_array([3,4,5])
```

Hybrid array:

```
| ?- g_assign(w,g_array([1,2,g_array([a,b,c]), g_array(2,z),5])), g_read(w, X).

X = g_array([1,2,g_array([a,b,c]), g_array([z,z]),5])
```

```
| ?- g_read(w(1), X), g_read(w(2,1), Y), g_read(w(3,1), Z).

X = 2
Y = b
Z = z

| ?- g_read(w(1,2),X).
uncaught exception: error(domain_error(g_array_index,w(1,2)),g_read/2)
```

Array extension:

```
| ?- g_assign(a, g_array([10,20,30])), g_read(a, X).

X = g_array([10,20,30])

| ?- g_assign(a, g_array_extend(5,null)), g_read(a, X).

X = g_array([10,20,30,null,null])

| ?- g_assign(a, g_array([10,20,30])), g_read(a, X).

X = g_array([10,20,30])

| ?- g_assign(a, g_array_extend([1,2,3,4,5,6])), g_read(a, X).

X = g_array([10,20,30,4,5,6])
```

Automatic array:

```
| ?- g_assign(t, g_array_auto(3)), g_assign(t(1), foo), g_read(t,X).

X = g_array([0,foo,0])

| ?- g_assign(t(5), bar), g_read(t,X).

X = g_array([0,foo,0,0,0,bar,0,0])

| ?- g_assign(t, g_array_auto(2, g_array(2))), g_assign(t(1,1), foo),
     g_read(t,X).

X = g_array([g_array([0,0]),g_array([0,foo])])

| ?- g_assign(t(3,0), bar), g_read(t,X).

X = g_array([g_array([0,0]),g_array([0,foo]),g_array([0,0]),g_array([bar,0])])

| ?- g_assign(t(3,4), bar), g_read(t,X).
uncaught exception: error(domain_error(g_array_index,t(3,4)),g_assign/2)

| ?- g_assign(t, g_array_auto(2, g_array_auto(2))), g_assign(t(1,1), foo),
     g_read(t,X).

X = g_array([g_array([0,0]),g_array([0,foo])])

| ?- g_assign(t(3,3), bar), g_read(t,X).

X = g_array([g_array([0,0]),g_array([0,foo]),g_array([0,0]),
    g_array([0,0,0,bar])])
```

```
| ?- g_assign(t, g_array_auto(2, g_array_auto(2, null))), g_read(t(2,3), U),
     g_read(t, X).

U = null
X = g_array([g_array([null,null]),g_array([null,null]),
             g_array([null,null,null,null]),g_array([null,null])])
```

7.22 Prolog state

7.22.1 set_prolog_flag/2

Templates

set_prolog_flag(+flag, +term)

Description

set_prolog_flag(Flag, Value) sets the value of the Prolog flag Flag to Value.

Prolog flags: a Prolog flag is an atom which is associated with a value that is either implementation defined or defined by the user. Each flag has a permitted range of values; any other value is a domain_error. The following two tables present available flags, the possible values, a description and if they are ISO or an extension. The first table presents unchangeable flags while the second one the changeable flags. For flags whose default values is machine independent, this value is <u>underlined</u>.

Unchangeable flags:

Flag	Values	Description	ISO
bounded	<u>true</u> / false	are integers bounded ?	Y
max_integer	an integer	greatest integer	Y
min_integer	an integer	smallest integer	Y
integer_rounding_function	toward_zero down	*rnd*(X) = integer part of X *rnd*(X) = $\lfloor X \rfloor$ (section 7.6.1, page 57)	Y
max_arity	an integer	maximum arity for compound terms (255)	Y
max_atom	an integer	maximum number of atoms	N
max_unget	an integer	maximum number of successive ungets	N
prolog_name	an atom	name of the Prolog system	N
prolog_version	an atom	version number of the Prolog system	N
prolog_date	an atom	date of the Prolog system	N
prolog_copyright	an atom	copyright message of the Prolog system	N

Changeable flags:

Flag	Values	Description	ISO
`char_conversion`	`on` / `off`	is character conversion activated ?	Y
`debug`	`on` / `off`	is the debugger activated ?	Y
`singleton_warning`	`on` / `off`	warn about named singleton variables ?	N
`strict_iso`	`on` / `off`	strict ISO behavior ?	N
		a double quoted constant is returned as:	
`double_quotes`	`atom`	an atom	Y
	`chars`	a list of characters	
	`codes`	a list of character codes	
	`atom_no_escape`	as `atom` but ignore escape sequences	N
	`chars_no_escape`	as `chars` but ignore escape sequences	
	`codes_no_escape`	as `code` but ignore escape sequences	
		a back quoted constant is returned as:	
`back_quotes`	`atom`	an atom	N
	`chars`	a list of characters	
	`codes`	a list of character codes	
	`atom_no_escape`	as `atom` but ignore escape sequences	
	`chars_no_escape`	as `chars` but ignore escape sequences	
	`codes_no_escape`	as `code` but ignore escape sequences	
		a predicate calls an unknown procedure:	
`unknown`	`error`	an `existence_error` is raised	Y
	`warning`	a message is displayed then fails	
	`fail`	quietly fails	
		a predicate causes a syntax error:	
`syntax_error`	`error`	a `syntax_error` is raised	N
	`warning`	a message is displayed then fails	
	`fail`	quietly fails	
		a predicate causes an O.S. error:	
`os_error`	`error`	a `system_error` is raised	N
	`warning`	a message is displayed then fails	
	`fail`	quietly fails	

The `strict_iso` flag is introduced to allow a compatibility with other Prolog systems. When turned off the following relaxations apply:

- a callable term can be given as a predicate indicator.
- built-in predicates are found by `current_predicate/1` (section 7.8.1, page 64).

Errors

`Flag` is a variable	`instantiation_error`
`Value` is a variable	`instantiation_error`
`Flag` is neither a variable nor an atom	`type_error(atom, Flag)`
`Flag` is an atom but not a valid flag	`domain_error(prolog_flag, Flag)`
`Value` is inappropriate for Flag	`domain_error(flag_value, Flag+Value)`
`Value` is appropriate for `Flag` but flag `Flag` is not modifiable	`permission_error(modify, flag, Flag)`

Portability

ISO predicate. All ISO flags are implemented.

7.22.2 `current_prolog_flag/2`

Templates

```
current_prolog_flag(?flag, ?term)
```

Description

`current_prolog_flag(Flag, Value)` succeeds if there exists a Prolog flag that unifies with `Flag` and whose value unifies with `Value`. This predicate is re-executable on backtracking.

Errors

`Flag` is neither a variable nor an atom	`type_error(atom, Flag)`
`Flag` is an atom but not a valid flag	`domain_error(prolog_flag, Flag)`

Portability

ISO predicate.

7.22.3 `set_bip_name/2`

Templates

```
set_bip_name(+atom, +arity)
```

Description

`set_bip_name(Functor, Arity)` initializes the context of the error (section 5.3.1, page 37) with `Functor` and `Arity` (if `Arity` <0 only `Functor` is significant).

Errors

`Functor` is a variable	`instantiation_error`
`Arity` is a variable	`instantiation_error`
`Functor` is neither a variable nor an atom	`type_error(atom, Functor)`
`Arity` is neither a variable nor an integer	`type_error(integer, Arity)`

Portability

GNU Prolog predicate.

7.22.4 `current_bip_name/2`

Templates

```
current_bip_name(?atom, ?arity)
```

Description

`current_bip_name(Functor, Arity)` succeeds if `Functor` and `Arity` correspond to the context of the error (section 5.3.1, page 37) (if `Arity` <0 only `Functor` is significant).

Errors

`Functor` is neither a variable nor an atom	`type_error(atom, Functor)`
`Arity` is neither a variable nor an integer	`type_error(integer, Arity)`

Portability

GNU Prolog predicate.

7.22.5 write_pl_state_file/1, read_pl_state_file/1

Templates

```
write_pl_state_file(+source_sink)
read_pl_state_file(+source_sink)
```

Description

`write_pl_state_file(FileName)` writes onto `FileName` all information that influences the parsing of a term (section 7.14, page 91). This allows a sub-process written in Prolog to read this file and then process any Prolog term as done by the parent process. This file can also be passed as argument of the `--pl-state` option when invoking `gplc` (section 3.4.3, page 22). More precisely the following elements are saved:

- all operator definitions (section 7.14.10, page 100).
- the character conversion table (section 7.14.12, page 102).
- the value of `char_conversion`, `double_quotes`, `back_quotes` and `singleton_warning` Prolog flags (section 7.22.1, page 133).

`read_pl_state_file(FileName)` reads (restores) from `FileName` all information previously saved by `write_pl_state_file/1`.

Errors

`FileName` is a variable	`instantiation_error`
`FileName` is neither a variable nor an atom	`type_error(atom, FileName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.23 Program state

7.23.1 consult/1, '.'/2 - program consult

Templates

```
consult(+atom_or_atom_list)
'.'(+atom, +atom_list)
```

Description

`consult(Files)` compiles and loads into memory each file of the list `Files`. Each file is compiled for byte-code using the GNU Prolog compiler (section 3.4, page 20) then loaded using `load/1` (section 7.23.2, page 137). It is possible to specify `user` as a file name to directly enter the program from the terminal. `Files` can be also a single file name (i.e. an atom). Refer to the section concerning the consult of a Prolog program for more information (section 3.2.3, page 16).

The final file name of a file is computed using the predicates `prolog_file_name/2` (section 7.26.3, page 142) and `absolute_file_name/2` (section 7.26.1, page 141).

`[ File | Files ]`, i.e. `'.'(File, Files)` is equivalent to `consult([ File | Files ])`.

Errors

`Files` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Files` is neither a partial list nor a list nor an atom	`type_error(list, Files)`
an element `E` of the `Files` list is neither a variable nor an atom	`type_error(atom, E)`
an element `E` of the `Files` list is an atom but not a valid pathname	`domain_error(os_path, E)`
an element `E` of the `Files` list is a valid pathname but does not correspond to an existing source	`existence_error(source_sink, E)`
an error occurs executing a directive	see `call/1` errors (section 6.2.3, page 47)

Portability

GNU Prolog predicates.

7.23.2 `load/1`

Templates

`load(+atom_or_atom_list)`

Description

`load(Files)` loads into memory each file of the list `Files`. Each file must have been previously compiled for byte-code using the GNU Prolog compiler (section 3.4, page 20). `Files` can be also a single file name (i.e. an atom).

The final file name of a file is computed using the predicates `absolute_file_name/2` (section 7.26.1, page 141). If no suffix is given `'.wbc'` is appended to the file name.

Errors

`Files` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Files` is neither a partial list nor a list nor an atom	`type_error(list, Files)`
an element `E` of the `Files` list is neither a variable nor an atom	`type_error(atom, E)`
an element `E` of the `Files` list is an atom but not a valid pathname	`domain_error(os_path, E)`
an element `E` of the `Files` list is a valid pathname but does not correspond to an existing source	`existence_error(source_sink, E)`
an error occurs executing a directive	see `call/1` errors (section 6.2.3, page 47)

Portability

GNU Prolog predicate.

7.23.3 `listing/1`, `listing/0`

Templates

```
listing(+predicate_indicator)
listing(+atom)
listing
```

Description

`listing(Pred)` lists the clauses of the consulted predicate whose predicate indicator is `Pred`. `Pred` can also be a single atom in which case all predicates whose name is `Pred` are listed (of any arity). This predicate uses `portray_clause/2` (section 7.14.8, page 99) to output the clauses.

`listing` lists all clauses of all consulted predicates.

Errors

`Pred` is a variable	`instantiation_error`
`Pred` is neither a variable nor predicate indicator or an atom	`type_error(predicate_indicator, Pred)`

Portability

GNU Prolog predicate.

7.24 System statistics

7.24.1 `statistics/0`, `statistics/2`

Templates

```
statistics
statistics(?atom, ?list)
```

Description

`statistics` displays statistics about memory usage and run times.

`statistics(Key, Value)` unifies `Value` with the current value of the statistics key `Key`. `Value` a list of two elements. Times are in milliseconds, sizes of areas in bytes.

Key	Description	Value
`user_time`	user CPU time	`[SinceStart, SinceLast]`
`system_time`	system CPU time	`[SinceStart, SinceLast]`
`cpu_time`	total CPU time (user + system)	`[SinceStart, SinceLast]`
`real_time`	absolute time	`[SinceStart, SinceLast]`
`local_stack`	local stack sizes (control, environments, choices)	`[UsedSize, FreeSize]`
`global_stack`	global stack sizes (compound terms)	`[UsedSize, FreeSize]`
`trail_stack`	trail stack sizes (variable bindings to undo)	`[UsedSize, FreeSize]`
`cstr_stack`	constraint trail sizes (finite domain constraints)	`[UsedSize, FreeSize]`

Note that the key `runtime` is recognized as `user_time` for compatibility purpose.

Errors

`Key` is neither a variable nor a valid key	`domain_error(statistics_key, Key)`
`Value` is neither a variable nor a list of two elements	`domain_error(statistics_value, Value)`
`Value` is a list of two elements and an element `E` is neither a variable nor an integer	`type_error(integer, E)`

Portability

GNU Prolog predicates.

7.24.2 `user_time/1`, `system_time/1`, `cpu_time/1`, `real_time/1`

Templates

```
user_time(?integer)
system_time(?integer)
cpu_time(?integer)
real_time(?integer)
```

Description

`user_time(Time)` unifies `Time` with the user CPU time elapsed since the start of Prolog.

`system_time(Time)` unifies `Time` with the system CPU time elapsed since the start of Prolog.

`cpu_time(Time)` unifies `Time` with the CPU time (user + system) elapsed since the start of Prolog.

`real_time(Time)` unifies `Time` with the absolute time elapsed since the start of Prolog.

Errors

`Time` is neither a variable nor an integer	`type_error(integer, Time)`

Portability

GNU Prolog predicates.

7.25 Random number generator

7.25.1 `set_seed/1`, `randomize/0`

Templates

```
set_seed(+integer)
randomize
```

Description

`set_seed(Seed)` reinitializes the random number generator seed with `Seed`.

`randomize` reinitializes the random number generator. This predicates calls `set_seed/1` with a random value depending on the absolute time.

Errors

`Seed` is a variable	`instantiation_error`
`Seed` is neither a variable nor an integer	`type_error(integer, Seed)`
`Seed` is an integer < 0	`domain_error(not_less_than_zero, Seed)`

Portability

GNU Prolog predicates.

7.25.2 `get_seed/1`

Templates

`get_seed(?integer)`

Description

`get_seed(Seed)` unifies `Seed` with the current random number generator seed.

Errors

`Seed` is neither a variable nor an integer	`type_error(integer, Seed)`
`Seed` is an integer < 0	`domain_error(not_less_than_zero, Seed)`

Portability

GNU Prolog predicate.

7.25.3 `random/1`

Templates

`random(-float)`

Description

`random(Number)` unifies `Number` with a random floating point number such that 0.0 ≤ `Number` < 1.0.

Errors

`Number` is not a variable	`type_error(variable, Number)`

Portability

GNU Prolog predicate.

7.25.4 `random/3`

Templates

`random(+number, +number, -number)`

Description

`random(Base, Max, Number)` unifies `Number` with a random number such that `Base` ≤ `Number` < `Max`. If both `Base` and `Max` are integers `Number` will be an integer, otherwise `Number` will be a floating point number.

Errors

`Base` is a variable	`instantiation_error`
`Base` is neither a variable nor a number	`type_error(number, Base)`
`Max` is a variable	`instantiation_error`
`Max` is neither a variable nor a number	`type_error(number, Max)`
`Number` is not a variable	`type_error(variable, Number)`

Portability

GNU Prolog predicate.

7.26 File name processing

7.26.1 `absolute_file_name/2`

Templates

`absolute_file_name(+atom, atom)`

Description

`absolute_file_name(File1, File2)` succeeds if `File2` is the absolute pathname associated to the relative file name `File1`. `File1` can contain `$`*`VAR_NAME`* sub-strings. When such a sub-string is encountered, it is expanded with the value of the environment variable whose name is *`VAR_NAME`* if exists (otherwise no expansion is done). `File1` can also begin with a sub-string `~`*`USER_NAME`*`/`, this is expanded as the home directory of the user *`USER_NAME`*. If *`USER_NAME`* does not exist `File1` is an invalid pathname. If no *`USER_NAME`* is given (i.e. `File1` begins with `~/`) the `~` character is expanded as the value of the environment variable `HOME`. If the `HOME` variable is not defined `File1` is an invalid pathname. Relative references to the current directory (`/./` sub-string) and to the parent directory (`/../` sub-strings) are removed and no longer appear in `File2`. `File1` is also invalid if it contains too many `/../` consecutive sub-strings (i.e. parent directory relative references). Finally if `File1` is `user` then `File2` is also unified with `user` to allow this predicate to be called on Prolog file names (since `user` in DEC-10 input/output predicates denotes the current input/output stream).

Most predicates using a file name implicitly call this predicate to obtain the desired file, e.g. `open/4`.

Errors

`File1` is a variable	`instantiation_error`
`File1` is neither a variable nor an atom	`type_error(atom, File1)`
`File2` is neither a variable nor an atom	`type_error(atom, File2)`
`File1` is an atom but not a valid pathname	`domain_error(os_path, File1)`

Portability

GNU Prolog predicate.

7.26.2 `decompose_file_name/4`

Templates

`decompose_file_name(+atom, ?atom, ?atom, ?atom)`

Description

`decompose_file_name(File, Directory, Prefix, Suffix)` decomposes the pathname `File` and extracts the `Directory` part (characters before the last `/`), the `Prefix` part (characters after the last `/` and before the last `.` or until the end if there is no suffix) and the `Suffix` part (characters from the last `.` to the end of the string).

Errors

`File` is a variable	`instantiation_error`
`File` is neither a variable nor an atom	`type_error(atom, File)`
`Directory` is neither a variable nor an atom	`type_error(atom, Directory)`
`Prefix` is neither a variable nor an atom	`type_error(atom, Prefix)`
`Suffix` is neither a variable nor an atom	`type_error(atom, Suffix)`

Portability

GNU Prolog predicate.

7.26.3 `prolog_file_name/2`

Templates

`prolog_file_name(+atom, ?atom)`

Description

`prolog_file_name(File1, File2)` unifies `File2` with the Prolog file name associated to `File1`. More precisely `File2` is computed as follows:

- if `File1` has a suffix or if it is `user` then `File2` is unified with `File1`.
- else if the file whose name is `File1` + `'.pl'` exists then `File2` is unified with this name.
- else if the file whose name is `File1` + `'.pro'` exists then `File2` is unified with this name.
- else `File2` is unified with the name `File1` + `'.pl'`.

This predicate uses `absolute_file_name/2` to check the existence of a file (section 7.26.1, page 141).

Errors

`File1` is a variable	`instantiation_error`
`File1` is neither a variable nor an atom	`type_error(atom, File1)`
`File2` is neither a variable nor an atom	`type_error(atom, File2)`
`File1` is an atom but not a valid pathname	`domain_error(os_path, File1)`

Portability

GNU Prolog predicate.

7.27 Operating system interface

7.27.1 `argument_counter/1`

Templates

`argument_counter(?integer)`

Description

`argument_counter(Counter)` succeeds if `Counter` is the number of arguments of the command-line. Since the first argument is always the name of the running program, `Counter` is always ≥ 1. See (section 3.2, page 13) for more information about command-line arguments retrieved under the top_level.

Errors

`Counter` is neither a variable nor an integer	`type_error(integer, Counter)`

Portability

GNU Prolog predicate.

7.27.2 `argument_value/2`

Templates

`argument_value(+integer, ?atom)`

Description

`argument_value(N, Arg)` succeeds if the `N`*th* argument on the command-line unifies with `Arg`. The first argument is always the name of the running program and its number is 0. The number of arguments on the command-line can be obtained using `argument_counter/1` (section 7.27.1, page 142).

Errors

`N` is a variable	`instantiation_error`
`N` is neither a variable nor an integer	`type_error(integer, N)`
`N` is an integer < 0	`domain_error(not_less_than_zero, N)`
`Arg` is neither a variable nor an atom	`type_error(atom, Arg)`

Portability

GNU Prolog predicate.

7.27.3 `argument_list/1`

Templates

`argument_list(?list)`

Description

`argument_list(Args)` succeeds if `Args` unifies with the list of atoms associated to each argument on the command-line other than the first argument (the name of the running program).

Errors

`Args` is neither a partial list nor a list	`type_error(list, Args)`

Portability

GNU Prolog predicate.

7.27.4 `environ/2`

Templates

```
environ(?atom, ?atom)
```

Description

`environ(Name, Value)` succeeds if `Name` is the name of an environment variable whose value is `Value`. This predicate is re-executable on backtracking.

Errors

`Name` is neither a variable nor an atom	`type_error(atom, Name)`
`Value` is neither a variable nor an atom	`type_error(atom, Value)`

Portability

GNU Prolog predicate.

7.27.5 `make_directory/1`, `delete_directory/1`, `change_directory/1`

Templates

```
make_directory(+atom)
delete_directory(+atom)
change_directory(+atom)
```

Description

`make_directory(PathName)` creates the directory whose pathname is `PathName`.

`delete_directory(PathName)` removes the directory whose pathname is `PathName`.

`change_directory(PathName)` sets the current directory to the directory whose pathname is `PathName`.

See `absolute_file_name/2` for information about the syntax of `PathName` (section 7.26.1, page 141).

Errors

`PathName` is a variable	`instantiation_error`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
`PathName` is an atom but not a valid pathname	`domain_error(os_path, PathName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*`atom explaining the error`*`)`

Portability

GNU Prolog predicates.

7.27.6 `working_directory/1`

Templates

`working_directory(?atom)`

Description

`working_directory(PathName)` succeeds if `PathName` is the pathname of the current directory.

Errors

`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`

Portability

GNU Prolog predicate.

7.27.7 `directory_files/2`

Templates

`directory_files(+atom, ?list)`

Description

`directory_files(PathName, Files)` succeeds if `Files` is the list of all entries (files, sub-directories,...) in the directory whose pathname is `PathName`. See `absolute_file_name/2` for information about the syntax of `PathName` (section 7.26.1, page 141).

Errors

`PathName` is a variable	`instantiation_error`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
`PathName` is an atom but not a valid pathname	`domain_error(os_path, PathName)`
`Files` is neither a partial list nor a list	`type_error(list, Files)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.8 `rename_file/2`

Templates

`rename_file(+atom, +atom)`

Description

`rename_file(PathName1, PathName2)` renames the file or directory whose pathname is `PathName1` to `PathName2`. See `absolute_file_name/2` for information about the syntax of `PathName1` and `PathName2` (section 7.26.1, page 141).

Errors

`PathName1` is a variable	`instantiation_error`
`PathName1` is neither a variable nor an atom	`type_error(atom, PathName1)`
`PathName1` is an atom but not a valid pathname	`domain_error(os_path, PathName1)`
`PathName2` is a variable	`instantiation_error`
`PathName2` is neither a variable nor an atom	`type_error(atom, PathName2)`
`PathName2` is an atom but not a valid pathname	`domain_error(os_path, PathName2)`
an operating system error occurs and value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.9 `delete_file/1`, `unlink/1`

Templates

```
delete_file(PathName)
unlink(PathName)
```

Description

`delete_file(PathName)` removes the existing file whose pathname is `PathName`.

`unlink/1` is similar to `delete_file/1` except that it never causes a `system_error` (e.g. if `PathName` does not refer to an existing file).

See `absolute_file_name/2` for information about the syntax of `PathName` (section 7.26.1, page 141).

Errors

`PathName` is a variable	`instantiation_error`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
`PathName` is an atom but not a valid pathname	`domain_error(os_path, PathName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.27.10 `file_permission/2`, `file_exists/1`

Templates

```
file_permission(+atom, +atom)
file_permission(+atom, +atom_list)
file_exists(+atom)
```

Description

`file_permission(PathName, Permission)` succeeds if `PathName` is the pathname of an existing file (or directory) whose permissions include `Permission`.

File permissions: `Permission` can be a single permission or a list of permissions. A permission is an atom among:

- `read`: the file or directory can be read.
- `write`: the file or directory can be written.
- `execute`: the file can be executed.
- `search`: the directory can be searched.

If `PathName` does not exists or if it its permissions do not include `Permission` this predicate fails.

`file_exists(PathName)` is equivalent to `file_permission(PathName, [])`, i.e. it succeeds if `PathName` is the pathname of an existing file (or directory).

See `absolute_file_name/2` for information about the syntax of `PathName` (section 7.26.1, page 141).

Errors

`PathName` is a variable	`instantiation_error`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
`PathName` is an atom but not a valid pathname	`domain_error(os_path, PathName)`
`Permission` is a partial list or a list with an element which is a variable	`instantiation_error`
`Permission` is neither an atom nor partial list or a list	`type_error(list, Permission)`
an element `E` of the `Permission` list is neither a variable nor an atom	`type_error(atom, E)`
an element `E` of the `Permission` is an atom but not a valid permission	`domain_error(os_file_permission, Permission)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.27.11 `file_property/2`

Templates

```
file_property(+atom, ?os_file_property)
```

Description

`file_property(PathName, Property)` succeeds if `PathName` is the pathname of an existing file (or directory) and if `Property` unifies with one of the properties of the file. This predicate is re-executable on backtracking.

File properties:

- `absolute_file_name(File)`: `File` is the absolute file name of `PathName` (section 7.26.1, page 141).
- `real_file_name(File)`: `File` is the real file name of `PathName` (follows symbolic links).
- `type(Type)`: `Type` is the type of `PathName`. Possible values are: `regular`, `directory`, `fifo`, `socket`, `character_device`, `block_device` or `unknown`.

- `size(Size)`: `Size` is the size (in bytes) of `PathName`.
- `permission(Permission)`: `Permission` is a permission of `PathName` (section 7.27.10, page 146).
- `last_modification(DT)`: `DT` is the last modification date and time (section 7.27.14, page 149).

See `absolute_file_name/2` for information about the syntax of `PathName` (section 7.26.1, page 141).

Errors

`PathName` is a variable	`instantiation_error`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
`PathName` is an atom but not a valid pathname	`domain_error(os_path, PathName)`
`Property` is neither a variable nor a file property term	`domain_error(os_file_property, Property)`
`Property` = `absolute_file_name(E)`, `real_file_name(E)`, `type(E)` or `permission(E)` and `E` is neither a variable nor an atom	`type_error(atom, E)`
`Property` = `last_modification(DateTime)` and `DateTime` is neither a variable nor a compound term	`type_error(compound, DateTime)`
`Property` = `last_modification(DateTime)` and `DateTime` is a compound term but not a structure `dt/6`	`domain_error(date_time, DateTime)`
`Property` = `size(E)` or `last_modification(DateTime)` and `DateTime` is a structure `dt/6` but an element `E` is neither a variable nor an integer	`type_error(integer, E)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.12 `temporary_name/2`

Templates

`temporary_name(+atom, ?atom)`

Description

`temporary_name(Template, PathName)` creates a unique file name `PathName` whose pathname begins by `Template`. `Template` should contain a pathname with six trailing `X`*s*. `PathName` is `Template` with the six `X`*s* replaced with a letter and the process identifier. This predicate is an interface to the C Unix function `mktemp(3)`.

See `absolute_file_name/2` for information about the syntax of `Template` (section 7.26.1, page 141).

Errors

`Template` is a variable	`instantiation_error`
`Template` is neither a variable nor an atom	`type_error(atom, Template)`
`Template` is an atom but not a valid pathname	`domain_error(os_path, Template)`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.13 `temporary_file/3`

Templates

```
temporary_file(+atom, +atom, ?atom)
```

Description

`temporary_file(Directory, Prefix, PathName)` creates a unique file name `PathName` whose pathname begins by `Directory/Prefix`. If `Directory` is the empty atom `''` a standard temporary directory will be used (e.g. `/tmp`). `Prefix` can be the empty atom `''`. This predicate is an interface to the C Unix function `tempnam(3)`.

See `absolute_file_name/2` for information about the syntax of `Directory` (section 7.26.1, page 141).

Errors

`Directory` is a variable	`instantiation_error`
`Directory` is neither a variable nor an atom	`type_error(atom, Directory)`
`Directory` is an atom but not a valid pathname	`domain_error(os_path, Directory)`
`Prefix` is a variable	`instantiation_error`
`Prefix` is neither a variable nor an atom	`type_error(atom, Prefix)`
`PathName` is neither a variable nor an atom	`type_error(atom, PathName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.14 `date_time/1`

Templates

```
date_time(?compound)
```

Description

`date_time(DateTime)` unifies `DateTime` with a compound term containing the current date and time. `DateTime` is a structure `dt(Year, Month, Day, Hour, Minute, Second)`. Each sub-argument of the term `dt/6` is an integer.

Errors

`DateTime` is neither a variable nor a compound term	`type_error(compound, DateTime)`
`DateTime` is a compound term but not a structure `dt/6`	`domain_error(date_time, DateTime)`
`DateTime` is a structure `dt/6` and an element `E` is neither a variable nor an integer	`type_error(integer, E)`

Portability

GNU Prolog predicate.

7.27.15 `host_name/1`

Templates

```
host_name(?atom)
```

Description

`host_name(HostName)` unifies `HostName` with the name of the host machine executing the current GNU Prolog process. If the sockets are available (section 7.28.1, page 157), the name returned will be fully qualified. In that case, `host_name/1` will also succeed if `HostName` is instantiated to the unqualified name (or an alias) of the machine.

Errors

`Hostname` is neither a variable nor an atom	`type_error(atom, HostName)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.16 `os_version/1`

Templates

```
os_version(?atom)
```

Description

`os_version(OSVersion)` unifies `OSVersion` with the operating system version of the machine executing the current GNU Prolog process.

Errors

`OSVersion` is neither a variable nor an atom	`type_error(atom, OSVersion)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.17 `architecture/1`

Templates

```
architecture(?atom)
```

Description

`architecture(Architecture)` unifies `Architecture` with the name of the machine executing the current GNU Prolog process.

Errors

`Architecture` is neither a variable nor an atom	`type_error(atom, Architecture)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*`atom explaining the error`*`)`

Portability

GNU Prolog predicate.

7.27.18 `shell/2`, `shell/1`, `shell/0`

Templates

```
shell(+atom, ?integer)
shell(+atom)
shell
```

Description

`shell(Command, Status)` invokes a new shell (named by the `SHELL` environment variable) passing `Command` for execution and unifies `Status` with the result of the execution. If `Command` is the empty atom `''` a new interactive shell is executed. The control is returned to Prolog upon termination of the called process.

`shell(Command)` is equivalent to `shell(Command, 0)`.

`shell` is equivalent to `shell('', 0)`.

Errors

`Command` is a variable	`instantiation_error`
`Command` is neither a variable nor an atom	`type_error(atom, Command)`
`Status` is neither a variable nor an integer	`type_error(integer, Status)`

Portability

GNU Prolog predicates.

7.27.19 system/2, system/1

Templates

```
system(+atom, ?integer)
system(+atom)
```

Description

`system(Command, Status)` invokes a new default shell passing `Command` for execution and unifies `Status` with the result of the execution. The control is returned to Prolog upon termination of the shell process. This predicate is an interface to the C Unix function `system(3)`.

`system(Command)` is equivalent to `system(Command, 0)`.

Errors

`Command` is a variable	`instantiation_error`
`Command` is neither a variable nor an atom	`type_error(atom, Command)`
`Status` is neither a variable nor an integer	`type_error(integer, Status)`

Portability

GNU Prolog predicates.

7.27.20 spawn/3, spawn/2

Templates

```
spawn(+atom, +atom_list, ?integer)
spawn(+atom, +atom_list)
```

Description

`spawn(Command, Arguments, Status)` executes `Command` passing as arguments of the command-line each element of the list `Arguments` and unifies `Status` with the result of the execution. The control is returned to Prolog upon termination of the command.

`spawn(Command, Arguments)` is equivalent to `spawn(Command, Arguments, 0)`.

Errors

`Command` is a variable	`instantiation_error`
`Command` is neither a variable nor an atom	`type_error(atom, Command)`
`Arguments` is a partial list or a list with an element which is a variable	`instantiation_error`
`Arguments` is neither a partial list nor a list	`type_error(list, Arguments)`
an element `E` of the `Arguments` list is neither a variable nor an atom	`type_error(atom, E)`
`Status` is neither a variable nor an integer	`type_error(integer, Status)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.27.21 popen/3

Templates

`popen(+atom, +io_mode, -stream)`

Description

`popen(Command, Mode, Stream)` invokes a new default shell (by creating a pipe) passing `Command` for execution and associates a stream either to the standard input or the standard output of the created process. if `Mode` is `read` (resp. `write`) an input (resp. output) stream is created and `Stream` is unified with the stream-term associated. Writing to the stream writes to the standard input of the command while reading from the stream reads the command's standard output. The stream must be closed using `close/2` (section 7.10.7, page 72). This predicate is an interface to the C Unix function `popen(3)`.

Errors

`Command` is a variable	`instantiation_error`
`Command` is neither a variable nor an atom	`type_error(atom, Command)`
`Mode` is a variable	`instantiation_error`
`Mode` is neither a variable nor an atom	`type_error(atom, Mode)`
`Mode` is an atom but neither `read` nor `write`.	`domain_error(io_mode, Mode)`
`Stream` is not a variable	`type_error(variable, Stream)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.22 exec/5, exec/4

Templates

```
exec(+atom, -stream, -stream, -stream, -integer)
exec(+atom, -stream, -stream, -stream)
```

Description

`exec(Command, StreamIn, StreamOut, StreamErr, Pid)` invokes a new default shell passing `Command` for execution and associates streams to standard streams of the created process. `StreamIn` is unified with the stream-term associated to the standard input stream of `Command` (it is an output stream). `StreamOut` is unified with the stream-term associated to the standard output stream of `Command` (it is an input stream). `StreamErr` is unified with the stream-term associated to the standard error stream of `Command` (it is an input stream). `Pid` is unified with the process identifier of the new process. This information is only useful if it is necessary to obtain the status of the execution using `wait/2` (section 7.27.25, page 155). Until a call to `wait/2` is done the process remains in the system processes table (as a zombie process if terminated). For this reason, if the status is not needed it is preferable to use `exec/4`.

`exec/4` is similar to `exec/5` but the process is removed from system processes as soon as it is terminated.

Errors

`Command` is a variable	`instantiation_error`
`Command` is neither a variable nor an atom	`type_error(atom, Command)`
`StreamIn` is not a variable	`type_error(variable, StreamIn)`
`StreamOut` is not a variable	`type_error(variable, StreamOut)`
`StreamErr` is not a variable	`type_error(variable, StreamErr)`
`Pid` is not a variable	`type_error(variable, Pid)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.27.23 `fork_prolog/1`

Templates

    fork_prolog(-integer)

Description

`fork_prolog(Pid)` creates a child process that differs from the parent process only in its PID. In the parent process `Pid` is unified with the PID of the child while in the child process `Pid` is unified with 0. In the parent process, the status of the child process can be ontained using `wait/2` (section 7.27.25, page 155). Until a call to `wait/2` is done the child process remains in the system processes table (as a zombie process if terminated). This predicate is an interface to the C Unix function `fork(2)`.

Errors

`Pid` is not a variable	`type_error(variable, Pid)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.24 `create_pipe/2`

Templates

    create_pipe(-stream, -stream)

Description

`create_pipe(StreamIn, StreamOut)` creates a pair of streams pointing to a pipe inode. `StreamIn` is unified with the stream-term associated to the input side of the pipe and `StreamOut` is unified with the stream-term associated to output side. This predicate is an interface to the C Unix function `pipe(2)`.

Errors

`StreamIn` is not a variable	`type_error(variable, StreamIn)`
`StreamOut` is not a variable	`type_error(variable, StreamOut)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.25 `wait/2`

Templates

```
wait(+integer, ?integer)
```

Description

`wait(Pid, Status)` waits for the child process whose identifier is `Pid` to terminate. `Status` is then unified with the exit status. This predicate is an interface to the C Unix function `waitpid(2)`.

Errors

`Pid` is a variable	`instantiation_error`
`Pid` is neither a variable nor an integer	`type_error(integer, Pid)`
`Status` is neither a variable nor an integer	`type_error(integer, Status)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.26 `prolog_pid/1`

Templates

```
prolog_pid(?integer)
```

Description

`prolog_pid(Pid)` unifies `Pid` with the process identifier of the current GNU Prolog process.

Errors

`Pid` is neither a variable nor an integer	`type_error(integer, Pid)`

Portability

GNU Prolog predicate.

7.27.27 send_signal/2

Templates

```
send_signal(+integer, +integer)
send_signal(+integer, +atom)
```

Description

`send_signal(Pid, Signal)` sends `Signal` to the process whose identifier is `Pid`. `Signal` can be specified directly as an integer or symbolically as an atom. Allowed atoms depend on the machine (e.g. `'SIGINT'`, `'SIGQUIT'`, `'SIGKILL'`, `'SIGUSR1'`, `'SIGUSR2'`, `'SIGALRM'`,...). This predicate is an interface to the C Unix function `kill(2)`.

Errors

`Pid` is a variable	`instantiation_error`
`Pid` is neither a variable nor an integer	`type_error(integer, Pid)`
`Signal` is a variable	`instantiation_error`
`Signal` is neither a variable nor an integer or an atom	`type_error(integer, Signal)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.27.28 sleep/1

Templates

```
sleep(+number)
```

Description

`sleep(Seconds)` puts the GNU Prolog process to sleep for `Seconds` seconds. `Seconds` can be an integer or a floating point number (in which case it can be < 1). This predicate is an interface to the C Unix function `usleep(3)`.

Errors

`Seconds` is a variable	`instantiation_error`
`Seconds` is neither a variable nor a number	`type_error(number, Seconds)`
`Seconds` is a number < 0	`domain_error(not_less_than_zero, Seconds)`

Portability

GNU Prolog predicate.

7.27.29 select/5

Templates

```
select(+list, ?list, +list, ?list, +number)
```

Description

`select(Reads, ReadyReads, Writes, ReadyWrites, TimeOut)` waits for a number of streams (or file descriptors) to change status. `ReadyReads` is unified with the list of elements listed in `Reads` that have characters available for reading. Similarly `ReadyWrites` is unified with the list of elements of `Writes` that are ok for immediate writing. The elements of `Reads` and `Writes` are either stream-terms or aliases or integers considered as file descriptors, e.g. for sockets (section 7.28, page 157). Streams that must be tested with `select/5` should not be buffered. This can be done at the opening using `open/4` (section 7.10.6, page 70) or later using `set_stream_buffering/2` (section 7.10.27, page 82). `TimeOut` is an upper bound on the amount of time (in milliseconds) elapsed before `select/5` returns. If `TimeOut` ≤ 0 (no timeout) `select/5` waits until something is available (either or reading or for writing) and thus can block indefinitely. This predicate is an interface to the C Unix function `select(2)`.

Errors

`Reads` (or `Writes`) is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Reads` is neither a partial list nor a list	`type_error(list, Reads)`
`Writes` is neither a partial list nor a list	`type_error(list, Writes)`
`ReadyReads` is neither a partial list nor a list	`type_error(list, ReadyReads)`
`ReadyWrites` is neither a partial list nor a list	`type_error(list, ReadyWrites)`
an element `E` of the `Reads` (or `Writes`) list is neither a stream-term or alias nor an integer	`domain_error(stream_or_alias, E)`
an element `E` of the `Reads` (or `Writes`) list is not a selectable item	`domain_error(selectable_item, E)`
an element `E` of the `Reads` (or `Writes`) list is an integer < 0	`domain_error(not_less_than_zero, E)`
an element `E` of the `Reads` (or `Writes`) list is a stream-tern or alias not associated with an open stream	`existence_error(stream, E)`
an element `E` of the `Reads` list is associated to an output stream	`permission_error(input, stream, E)`
an element `E` of the `Writes` list is associated to an input stream	`permission_error(output, stream, E)`
`TimeOut` is a variable	`instantiation_error`
`TimeOut` is neither a variable nor a number	`type_error(number, TimeOut)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28 Sockets input/output

7.28.1 Introduction

This set of predicates provides a way to manipulate sockets. The predicates are straightforward interfaces to the corresponding BSD-type socket functions. This facility is available if the sockets part of GNU Prolog has been installed. A reader familiar with BSD sockets will understand them immediately otherwise a study of sockets is needed.

The domain is either the atom `'AF_INET'` or `'AF_UNIX'` corresponding to the same domains in BSD-type sockets.

An address is either of the form `'AF_INET'(HostName, Port)` or `'AF_UNIX'(SocketName)`. `HostName` is an atom denoting a machine name, `Port` is a port number and `SocketName` is an atom denoting a socket.

By default, streams associated to sockets are `block` buffered. The predicate `set_stream_buffering/2` (section 7.10.27, page 82) can be used to change this mode. They are also text streams by default. Use `set_stream_type/2` (section 7.10.25, page 81) to change the type if binary streams are needed.

7.28.2 `socket/2`

Templates

`socket(+socket_domain, -integer)`

Description

`socket(Domain, Socket)` creates a socket whose domain is `Domain` (section 7.28, page 157) and unifies `Socket` with the descriptor identifying the socket. This predicate is an interface to the C Unix function `socket(2)`.

Errors

`Domain` is a variable	`instantiation_error`
`Domain` is neither a variable nor an atom	`type_error(atom, Domain)`
`Domain` is an atom but not a valid socket domain	`domain_error(socket_domain, Domain)`
`Socket` is not a variable	`type_error(variable, Socket)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28.3 `socket_close/1`

Templates

`socket_close(+integer)`

Description

`socket_close(Socket)` closes the socket whose descriptor is `Socket`. This predicate should not be used if `Socket` has given rise to a stream, e.g. by `socket_connect/4` (section 7.28.5, page 159). In that case simply use `close/2` (section 7.10.7, page 72) on the associated stream.

Errors

`Socket` is a variable	`instantiation_error`
`Socket` is neither a variable nor an integer	`type_error(integer, Socket)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28.4 `socket_bind/2`

Templates

```
socket_bind(+integer, +socket_address)
```

Description

`socket_bind(Socket, Address)` binds the socket whose descriptor is `Socket` to the address specified by `Address` (section 7.28, page 157). If `Address` if of the form `'AF_INET'(HostName, Port)` and if `HostName` is uninstantiated then it is unified with the current machine name. If `Port` is uninstantiated, it is unified to a port number picked by the operating system. This predicate is an interface to the C Unix function `bind(2)`.

Errors

`Socket` is a variable	`instantiation_error`
`Socket` is neither a variable nor an integer	`type_error(integer, Socket)`
`Address` is a variable	`instantiation_error`
`Address` is neither a variable nor a valid address	`domain_error(socket_address, Address)`
`Address` = `'AF_UNIX'(E)` and `E` is a variable	`instantiation_error`
`Address` = `'AF_UNIX'(E)` or `'AF_INET'(E, _)` and `E` is neither a variable nor an atom	`type_error(atom, E)`
`Address` = `'AF_UNIX'(E)` and `E` is an atom but not a valid pathname	`domain_error(os_path, E)`
`Address` = `'AF_INET'(_, E)` and `E` is neither a variable nor an integer	`type_error(integer, E)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28.5 `socket_connect/4`

Templates

```
socket_connect(+integer, +socket_address, -stream, -stream)
```

Description

`socket_connect(Socket, Address, StreamIn, StreamOut)` connects the socket whose descriptor is `Socket` to the address specified by `Address` (section 7.28, page 157). `StreamIn` is unified with a stream-term associated to the input of the connection (it is an input stream). Reading from this stream gets data from the socket. `StreamOut` is unified with a stream-term associated to the output of the connection (it is an output stream). Writing to this stream sends data to the socket. The use of `select/5` can be useful (section 7.27.29, page 156). This predicate is an interface to the C Unix function `connect(2)`.

Errors

`Socket` is a variable	`instantiation_error`
`Socket` is neither a variable nor an integer	`type_error(integer, Socket)`
`Address` is a variable	`instantiation_error`
`Address` is neither a variable nor a valid address	`domain_error(socket_address, Address)`
`Address` = `'AF_UNIX'(E)` or `'AF_INET'(E, _)` or `Address` = `'AF_INET'(_, E)` and `E` is a variable	`instantiation_error`
`Address` = `'AF_UNIX'(E)` or `'AF_INET'(E, _)` and `E` is neither a variable nor an atom	`type_error(atom, E)`
`Address` = `'AF_UNIX'(E)` and `E` is an atom but not a valid pathname	`domain_error(os_path, E)`
`Address` = `'AF_INET'(_, E)` and `E` is neither a variable nor an integer	`type_error(integer, E)`
`StreamIn` is not a variable	`type_error(variable, StreamIn)`
`StreamOut` is not a variable	`type_error(variable, StreamOut)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28.6 `socket_listen/2`

Templates

```
socket_listen(+integer, +integer)
```

Description

`socket_listen(Socket, Length)` defines the socket whose descriptor is `Socket` to have a maximum backlog queue of `Length` pending connections. This predicate is an interface to the C Unix function `listen(2)`.

Errors

`Socket` is a variable	`instantiation_error`
`Socket` is neither a variable nor an integer	`type_error(integer, Socket)`
`Length` is a variable	`instantiation_error`
`Length` is neither a variable nor an integer	`type_error(integer, Length)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicate.

7.28.7 `socket_accept/4`, `socket_accept/3`

Templates

```
socket_accept(+integer, -atom, -stream, -stream)
socket_accept(+integer, -stream, -stream)
```

Description

`socket_accept(Socket, Client, StreamIn, StreamOut)` extracts the first connection to the socket whose descriptor is `Socket`. If the domain is `'AF_INET'`, `Client` is unified with an atom whose name is the Internet host address in numbers-and-dots notation of the connecting machine. `StreamIn` is unified with a stream-term associated to the input of the connection (it is an input stream). Reading from this stream gets data from the socket. `StreamOut` is unified with a stream-term associated to the output of the connection (it is an output stream). Writing to this stream sends data to the socket. The use of `select/5` can be useful (section 7.27.29, page 156). This predicate is an interface to the C Unix function `accept(2)`.

`socket_accept(Socket, StreamIn, StreamOut)` is equivalent to `socket_accept(Socket, _, StreamIn, StreamOut)`.

Errors

`Socket` is a variable	`instantiation_error`
`Socket` is neither a variable nor an integer	`type_error(integer, Socket)`
`Client` is not a variable	`type_error(variable, Client)`
`StreamIn` is not a variable	`type_error(variable, StreamIn)`
`StreamOut` is not a variable	`type_error(variable, StreamOut)`
an operating system error occurs and the value of the `os_error` Prolog flag is `error` (section 7.22.1, page 133)	`system_error(`*atom explaining the error*`)`

Portability

GNU Prolog predicates.

7.28.8 `hostname_address/2`

Templates

```
hostname_address(+atom, ?atom)
hostname_address(?atom, +atom)
```

Description

`hostname_address(HostName, HostAddress)` succeeds if the Internet host address in numbers-and-dots notation of `HostName` is `HostAddress`. `Hostname` can be given as a fully qualified name, or an unqualified name or an alias of the machine. The predicate will fail if the machine name or address cannot be resolved.

Errors

`HostName` and `HostAddress` are variables	`instantiation_error`
`HostName` is neither a variable nor an atom	`type_error(atom, HostName)`
`HostAddress` is neither a variable nor an atom	`type_error(atom, HostAddress)`
`Address` is neither a variable nor a valid address	`domain_error(socket_address, Address)`

Portability

GNU Prolog predicate.

7.29 Linedit management

The following predicates are only available if the `linedit` part of GNU Prolog has been installed.

7.29.1 `get_linedit_prompt/1`

Templates

`get_linedit_prompt(?atom)`

Description

`get_linedit_prompt(Prompt)` succeeds if `Prompt` is the current `linedit` prompt, e.g. the top-level prompt is `'| ?-'`. By default all other reads have an empty prompt.

Errors

`Prompt` is neither a variable nor an atom	`type_error(atom, Pred)`

Portability

GNU Prolog predicate.

7.29.2 `set_linedit_prompt/1`

Templates

`set_linedit_prompt(+atom)`

Description

`set_linedit_prompt(Prompt)` sets the current `linedit` prompt to `Prompt`. This prompt will be displayed for reads from a terminal (except for top-level reads).

Errors

`Prompt` is a variable	`instantiation_error`
`Prompt` is neither a variable nor an atom	`type_error(atom, Pred)`

Portability

GNU Prolog predicate.

7.29.3 `add_linedit_completion/1`

Templates

`add_linedit_completion(+atom)`

Description

`add_linedit_completion(Word)` adds `Word` in the list of completion words maintained by `linedit` (section 3.2.5, page 18). Only words containing letters, digits and the underscore character are added (if `Word` does not respect this restriction the predicate fails).

Errors

`Word` is a variable	`instantiation_error`
`Word` is neither a variable nor an atom	`type_error(atom, Word)`

Portability

GNU Prolog predicate.

7.29.4 `find_linedit_completion/2`

Templates

`find_linedit_completion(+atom, ?atom)`

Description

`find_linedit_completion(Prefix, Word)` succeeds if `Word` is a word beginning by `Prefix` and belongs to the list of completion words maintained by `linedit` (section 3.2.5, page 18). This predicate is re-executable on backtracking.

Errors

`Prefix` is a variable	`instantiation_error`
`Prefix` is neither a variable nor an atom	`type_error(atom, Prefix)`
`Word` is neither a variable nor an atom	`type_error(atom, Word)`

Portability

GNU Prolog predicate.

7.30 Source reader facility

7.30.1 Introduction

To be written...

7.30.2 sr_open/3

7.30.3 sr_change_options/2

7.30.4 sr_close/1

7.30.5 sr_read_term/4

7.30.6 sr_current_descriptor/1

7.30.7 sr_get_stream/2

7.30.8 sr_get_module/3

7.30.9 sr_get_file_name/2

7.30.10 sr_get_position/3

7.30.11 sr_get_include_list/2

7.30.12 sr_get_include_stream_list/2

7.30.13 sr_get_size_counters/3

7.30.14 sr_get_error_counters/3

7.30.15 sr_set_error_counters/3

7.30.16 sr_error_from_exception/2

7.30.17 sr_write_message/8, sr_write_message/6, sr_write_message/4

7.30.18 sr_write_error/6, sr_write_error/4, sr_write_error/2

8 Finite domain solver and built-in predicates

8.1 Introduction

The finite domain (FD) constraint solver extends Prolog with constraints over FD. This facility is available if the FD part of GNU Prolog has been installed. The solver is an instance of the Constraint Logic Programming scheme introduced by Jaffar and Lassez in 1987 [6]. Constraints on FD are solved using propagation techniques, in particular arc-consistency (AC). The interested reader can refer to "Constraint Satisfaction in Logic Programming" of P. Van Hentenryck (1989) [7]. The solver is based on the `clp(FD)` solver [4]. The GNU Prolog FD solver offers arithmetic constraints, boolean constraints, reified constraints and symbolic constraints on an new kind of variables: Finite Domain variables.

8.1.1 Finite Domain variables

A new type of data is introduced: FD variables which can only take values in their domains. The initial domain of an FD variable is `0..fd_max_integer` where `fd_max_integer` represents the greatest value that any FD variable can take. The predicate `fd_max_integer/1` returns this value which may be different from the `max_integer` Prolog flag (section 7.22.1, page 133). The domain of an FD variable `X` is reduced step by step by constraints in a monotonic way: when a value has been removed from the domain of `X` it will never reappear in the domain of `X`. An FD variable is fully compatible with both Prolog integers and Prolog variables. Namely, when an FD variable is expected by an FD constraint it is possible to pass a Prolog integer (considered as an FD variable whose domain is a singleton) or a Prolog variable (immediately bound to an initial range `0..fd_max_integer`). This avoids the need for specific type declaration. Although it is not necessary to declare the initial domain of an FD variable (since it will be bound `0..fd_max_integer` when appearing for the fist time in a constraint) it is advantageous to do so and thus reduce as soon as possible the size of its domain: particularly because GNU Prolog, for efficiency reasons, does not check for overflows. For instance, without any preliminary domain definitions for `X`, `Y` and `Z`, the non-linear constraint `X*Y#=Z` will fail due to an overflow when computing the upper bound of the domain of `Z`: `fd_max_integer` × `fd_max_integer`. This overflow causes a negative result for the upper bound and the constraint then fails.

There are two internal representations for an FD variable:

- **interval representation**: only the *min* and the *max* of the variable are maintained. In this representation it is possible to store values included in `0..fd_max_integer`.
- **sparse representation**: an additional bit-vector is used to store the set of possible values for the variable (i.e. the domain). In this representation it is possible to store values included in `0..vector_max`. By default `vector_max` is set to 127. This value can be redefined via an environment variable `VECTORMAX` or via the built-in predicate `fd_set_vector_max/1` (section 8.2.3, page 167). The predicate `fd_vector_max/1` returns the current value of `vector_max` (section 8.2.1, page 166).

The initial representation for an FD variable `X` is always an interval representation and is switched to a sparse representation when a "hole" appears in the domain (e.g. due to an inequality constraint). Once a variable uses a sparse representation it will not switch back to an interval representation even if there are no longer holes in its domain. When this switching occurs some values in the domain of `X` can be lost since `vector_max` is less than `fd_max_integer`. We say that "`X` is extra-constrained" since `X` is constrained by the solver to the domain `0..vector_max` (via an imaginary constraint `X #=<` *`vector_max`*). An `extra_cstr` is associated to each FD variable to indicate that values have been lost due to the switch to a sparse representation. This flag is updated on every operations. The domain of an extra-constrained FD variable is output followed by the `@` symbol. When a constraint fails on a extra-constrained variable a message `Warning: Vector too small - maybe lost solutions (FD Var:`*`N`*`)` is displayed (*N* is the address of the involved variable).

Example 1 (`vector_max` = `127`):

Constraint on `X`	Domain of `X`	`extra_cstr`	Lost values
`X #=< 512`	`0..512`	`off`	none
`X #\= 10`	`0..9:11..127`	`on`	`128..512`
`X #=< 100`	`0..9:11..100`	`off`	none

In this example, when the constraint `X #\= 10` is posted some values are lost, the `extra_cstr` is then switched on. However, posting the constraint `X #=< 100` will turn off the flag (no values are lost).

Example 2:

Constraint on `X`	Domain of `X`	`extra_cstr`	Lost values
`X #=< 512`	`0..512`	`off`	none
`X #\= 10`	`0..9:11..127`	`on`	`128..512`
`X #>= 256`	`Warning: Vector too small...`	`on`	`128..512`

In this example, the constraint `X #>= 256` fails due to the lost of `128..512` so a message is displayed onto the terminal. The solution would consist in increasing the size of the vector either by setting the environment variable `VECTORMAX` (e.g. to `512`) or using `fd_set_vector_max(512)`.

Finally, bit-vectors are not dynamic, i.e. all vectors have the same size (`0..vector_max`). So the use of `fd_set_vector_max/1` is limited to the initial definition of vector sizes and must occur before any constraint. As seen before, the solver tries to display a message when a failure occurs due to a too short `vector_max`. Unfortunately, in some cases it cannot detect the lost of values and no message is emitted. So the user should always take care to this parameter to be sure that it is large to encode any vector.

8.2 FD variable parameters

8.2.1 `fd_max_integer/1`

Templates

`fd_max_integer(?integer)`

Description

`fd_max_integer(N)` succeeds if `N` is the current value of `fd_max_integer` (section 8.1, page 165).

Errors

`N` is neither a variable nor an integer	`type_error(integer, N)`

Portability

GNU Prolog predicate.

8.2.2 `fd_vector_max/1`

Templates

`fd_vector_max(?integer)`

Description

`fd_vector_max(N)` succeeds if `N` is the current value of `vector_max` (section 8.1, page 165).

Errors

`N` is neither a variable nor an integer	`type_error(integer, N)`

Portability

GNU Prolog predicate.

8.2.3 `fd_set_vector_max/1`

Templates

```
fd_set_vector_max(+integer)
```

Description

`fd_set_vector_max(N)` initializes `vector_max` based on the value `N` (section 8.1, page 165). More precisely, on 32 bit machines, `vector_max` is set to the smallest value of `(32*k)-1` which is $\geq$ `N`.

Errors

`N` is a variable	`instantiation_error`
`N` is neither a variable nor an integer	`type_error(integer, N)`
`N` is an integer < 0	`domain_error(not_less_than_zero, N)`

Portability

GNU Prolog predicate.

8.3 Initial value constraints

8.3.1 `fd_domain/3`, `fd_domain_bool/1`

Templates

```
fd_domain(+fd_variable_list_or_fd_variable, +integer, +integer)
fd_domain(?fd_variable, +integer, +integer)
fd_domain_bool(+fd_variable_list)
fd_domain_bool(?fd_variable)
```

Description

`fd_domain(Vars, Lower, Upper)` constraints each element `X` of `Vars` to take a value in `Lower..Upper`. This predicate is generally used to set the initial domain of variables to an interval. `Vars` can be also a single FD variable (or a single Prolog variable).

`fd_domain_bool(Vars)` is equivalent to `fd_domain(Vars, 0, 1)` and is used to declare boolean FD variables.

Errors

`Vars` is not a variable but is a partial list	`instantiation_error`
`Vars` is neither a variable nor an FD variable nor an integer nor a list	`type_error(list, Vars)`
an element `E` of the `Vars` list is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, E)`
`Lower` is a variable	`instantiation_error`
`Lower` is neither a variable nor an integer	`type_error(integer, Lower)`
`Upper` is a variable	`instantiation_error`
`Upper` is neither a variable nor an integer	`type_error(integer, Upper)`

Portability

GNU Prolog predicate.

8.3.2 `fd_domain/2`

Templates

```
fd_domain(+fd_variable_list, +integer_list)
fd_domain(?fd_variable, +integer_list)
```

Description

`fd_domain(Vars, Values)` constraints each element `X` of the list `Vars` to take a value in the list `Values`. This predicate is generally used to set the initial domain of variables to a set of values. The domain of each variable of `Vars` uses a sparse representation. `Vars` can be also a single FD variable (or a single Prolog variable).

Errors

`Vars` is not a variable but is a partial list	`instantiation_error`
`Vars` is neither a variable nor an FD variable nor an integer nor a list	`type_error(list, Vars)`
an element `E` of the `Vars` list is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, E)`
`Values` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Values` is neither a partial list nor a list	`type_error(list, Values)`
an element `E` of the `Values` list is neither a variable nor an integer	`type_error(integer, E)`

Portability

GNU Prolog predicate.

8.4 Type testing

8.4.1 `fd_var/1`, `non_fd_var/1`, `generic_var/1`, `non_generic_var/1`

Templates

```
fd_var(?term)
non_fd_var(?term)
generic_var(?term)
non_generic_var(?term)
```

Description

`fd_var(Term)` succeeds if `Term` is currently an FD variable.

`non_fd_var(Term)` succeeds if `Term` is currently not an FD variable (opposite of `fd_var/1`).

`generic_var(Term)` succeeds if `Term` is either a Prolog variable or an FD variable.

`non_generic_var(Term)` succeeds if `Term` is neither a Prolog variable nor an FD variable (opposite of `generic_var/1`).

Errors

None.

Portability

GNU Prolog predicate.

8.5 FD variable information

These predicate allow the user to get some information about FD variables. They are not constraints, they only return the current state of a variable.

8.5.1 `fd_min/2`, `fd_max/2`, `fd_size/2`, `fd_dom/2`

Templates

```
fd_min(+fd_variable, ?integer)
fd_max(+fd_variable, ?integer)
fd_size(+fd_variable, ?integer)
fd_dom(+fd_variable, ?integer_list)
```

Description

`fd_min(X, N)` succeeds if `N` is the minimal value of the current domain of `X`.

`fd_max(X, N)` succeeds if `N` is the maximal value of the current domain of `X`.

`fd_size(X, N)` succeeds if `N` is the number of elements of the current domain of `X`.

`fd_dom(X, Values)` succeeds if `Values` is the list of values of the current domain of `X`.

Errors

`X` is a variable	`instantiation_error`
`X` is neither an FD variable nor an integer	`type_error(fd_variable, X)`
`N` is neither a variable nor an integer	`type_error(integer, N)`
an element `E` of the `Vars` list is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, E)`
`Values` is neither a partial list nor a list	`type_error(list, Values)`

Portability

GNU Prolog predicate.

8.5.2 fd_has_extra_cstr/1, fd_has_vector/1, fd_use_vector/1

Templates

```
fd_has_extra_cstr(+fd_variable)
fd_has_vector(+fd_variable)
fd_use_vector(+fd_variable)
```

Description

`fd_has_extra_cstr(X)` succeeds if the `extra_cstr` of `X` is currently on (section 8.1, page 165).

`fd_has_vector(X)` succeeds if the current domain of `X` uses a sparse representation (section 8.1, page 165).

`fd_use_vector(X)` enforces a sparse representation for the domain of `X` (section 8.1, page 165).

Errors

`X` is a variable	`instantiation_error`
`X` is neither an FD variable nor an integer	`type_error(fd_variable, X)`

Portability

GNU Prolog predicates.

8.6 Arithmetic constraints

8.6.1 FD arithmetic expressions

An FD arithmetic expression is a Prolog term built from integers, variables (Prolog or FD variables), and functors (or operators) that represent arithmetic functions. The following table details the components of an FD arithmetic expression:

FD Expression	Result
Prolog variable	domain `0..fd_max_integer`
FD variable `X`	domain of `X`
integer number `N`	domain `N..N`
`+ E`	same as `E`
`- E`	opposite of `E`
`E1 + E2`	sum of `E1` and `E2`
`E1 - E2`	subtraction of `E2` from `E1`
`E1 * E2`	multiplication of `E1` by `E2`
`E1 / E2`	integer division of `E1` by `E2` (only succeeds if the remainder is 0)
`E1 ** E2`	`E1` raised to the power of `E2` (`E1` or `E2` must be an integer)
`min(E1,E2)`	minimum of `E1` and `E2`
`max(E1,E2)`	maximum of `E1` and `E2`
`dist(E1,E2)`	distance, i.e. |`E1 - E2`|
`E1 // E2`	quotient of the integer division of `E1` by `E2`
`E1 rem E2`	remainder of the integer division of `E1` by `E2`
`quot_rem(E1,E2,R)`	quotient of the integer division of `E1` by `E2` (`R` is the remainder of the integer division of `E1` by `E2`)

FD expressions are not restricted to be linear. However non-linear constraints usually yield less constraint propagation than linear constraints.

`+`, `-`, `*`, `/`, `//`, `rem` and `**` are predefined infix operators. `+` and `-` are predefined prefix operators (section 7.14.10, page 100).

Errors

a sub-expression is of the form `_ ** E` and `E` is a variable	`instantiation_error`
a sub-expression `E` is neither a variable nor an integer nor an FD arithmetic functor	`type_error(fd_evaluable, E)`
an expression is too complex	`resource_error(too_big_fd_constraint)`

8.6.2 Partial AC: `(#=)/2` - constraint equal, `(#\=)/2` - constraint not equal, `(#<)/2` - constraint less than, `(#=<)/2` - constraint less than or equal, `(#>)/2` - constraint greater than, `(#>=)/2` - constraint greater than or equal

Templates

```
#=(?fd_evaluable, ?fd_evaluable)
#\=(?fd_evaluable, ?fd_evaluable)
#<(?fd_evaluable, ?fd_evaluable)
#=<(?fd_evaluable, ?fd_evaluable)
#>(?fd_evaluable, ?fd_evaluable)
#>=(?fd_evaluable, ?fd_evaluable)
```

Description

`FdExpr1 #= FdExpr2` constrains `FdExpr1` to be equal to `FdExpr2`.

`FdExpr1 #\= FdExpr2` constrains `FdExpr1` to be different from `FdExpr2`.

`FdExpr1 #< FdExpr2` constrains `FdExpr1` to be less than `FdExpr2`.

`FdExpr1 #=< FdExpr2` constrains `FdExpr1` to be less than or equal to `FdExpr2`.

`FdExpr1 #> FdExpr2` constrains `FdExpr1` to be greater than `FdExpr2`.

`FdExpr1 #>= FdExpr2` constrains `FdExpr1` to be greater than or equal to `FdExpr2`.

`FdExpr1` and `FdExpr2` are arithmetic FD expressions (section 8.6.1, page 170).

`#=`, `#\=`, `#<`, `#=<`, `#>` and `#>=` are predefined infix operators (section 7.14.10, page 100).

These predicates post arithmetic constraints that are managed by the solver using a partial arc-consistency algorithm to reduce the domain of involved variables. In this scheme only the bounds of the domain of variables are updated. This leads to less propagation than full arc-consistency techniques (section 8.6.3, page 172) but is generally more efficient for arithmetic. These arithmetic constraints can be reified (section 8.7, page 173).

Errors

Refer to the syntax of arithmetic FD expressions for possible errors (section 8.6.1, page 170).

Portability

GNU Prolog predicates.

8.6.3 Full AC: (#=#)/2 - constraint equal, (#\=#)/2 - constraint not equal, (#<#)/2 - constraint less than, (#=<#)/2 - constraint less than or equal, (#>#)/2 - constraint greater than, (#>=#)/2 - constraint greater than or equal

Templates

```
#=#(?fd_evaluable, ?fd_evaluable)
#\=#(?fd_evaluable, ?fd_evaluable)
#<#(?fd_evaluable, ?fd_evaluable)
#=<#(?fd_evaluable, ?fd_evaluable)
#>#(?fd_evaluable, ?fd_evaluable)
#>=#(?fd_evaluable, ?fd_evaluable)
```

Description

`FdExpr1 #=# FdExpr2` constrains `FdExpr1` to be equal to `FdExpr2`.

`FdExpr1 #\=# FdExpr2` constrains `FdExpr1` to be different from `FdExpr2`.

`FdExpr1 #<# FdExpr2` constrains `FdExpr1` to be less than `FdExpr2`.

`FdExpr1 #=<# FdExpr2` constrains `FdExpr1` to be less than or equal to `FdExpr2`.

`FdExpr1 #>#  FdExpr2` constrains `FdExpr1` to be greater than `FdExpr2`.

`FdExpr1 #>=# FdExpr2` constrains `FdExpr1` to be greater than or equal to `FdExpr2`.

`FdExpr1` and `FdExpr2` are arithmetic FD expressions (section 8.6.1, page 170).

`#=#`, `#\=#`, `#<#`, `#=<#`, `#>#` and `#>=#` are predefined infix operators (section 7.14.10, page 100).

These predicates post arithmetic constraints that are managed by the solver using a full arc-consistency algorithm to reduce the domain of involved variables. In this scheme the full domain of variables is updated. This leads to more propagation than partial arc-consistency techniques (section 8.6.1, page 170) but is generally less efficient for arithmetic. These arithmetic constraints can be reified (section 8.7.1, page 173).

Errors

Refer to the syntax of arithmetic FD expressions for possible errors (section 8.6.1, page 170).

Portability

GNU Prolog predicates.

8.6.4 `fd_prime/1`, `fd_not_prime/1`

Templates

```
fd_prime(?fd_variable)
fd_not_prime(?fd_variable)
```

Description

`fd_prime(X)` constraints `X` to be a prime number between `0..vector_max`. This constraint enforces a sparse representation for the domain of `X` (section 8.1, page 165).

`fd_not_prime(X)` constraints `X` to be a non prime number between `0..vector_max`. This constraint enforces a sparse representation for the domain of `X` (section 8.1, page 165).

Errors

`X` is neither an FD variable nor an integer	`type_error(fd_variable, X)`

Portability

GNU Prolog predicates.

8.7 Boolean and reified constraints

8.7.1 Boolean FD expressions

An boolean FD expression is a Prolog term built from integers (0 for false, 1 for true), variables (Prolog or FD variables), partial AC arithmetic constraints (section 8.6.2, page 171), full AC arithmetic constraints (section 8.6.3, page 172) and functors (or operators) that represent boolean functions. When a sub-expression of a boolean expression is an arithmetic constraint `c`, it is reified. Namely, as soon as the solver detects that `c` is true (i.e. *entailed*) the sub-expression has the value `1`. Similarly when the solver detects that `c` is false (i.e. *disentailed*) the sub-expression evaluates as `0`. While neither the entailment nor the disentailment can be detected the sub-expression is evaluated as a domain `0..1`. The following table details the components of an FD boolean expression:

FD Expression	Result
Prolog variable	domain `0..1`
FD variable `X`	domain of `X`, `X` is constrained to be in `0..1`
`0` (integer)	`0` (false)
`1` (integer)	`1` (true)
`#\ E`	not `E`
`E1 #<=> E2`	`E1` equivalent to `E2`
`E1 #\<=> E2`	`E1` not equivalent to `E2` (i.e. `E1` different from `E2`)
`E1 ## E2`	`E1` exclusive OR `E2` (i.e. `E1` not equivalent to `E2`)
`E1 #==> E2`	`E1` implies `E2`
`E1 #\==> E2`	`E1` does not imply `E2`
`E1 #/\ E2`	`E1` AND `E2`
`E1 #\/\ E2`	`E1` NAND `E2`
`E1 #\/ E2`	`E1` OR `E2`
`E1 #\\/ E2`	`E1` NOR `E2`

`#<=>`, `#\<=>`, `##`, `#==>`, `#\==>`, `#/\`, `#\/\`, `#\/` and `#\\/` are predefined infix operators. `#\` is a predefined prefix operator (section 7.14.10, page 100).

Errors

a sub-expression `E` is neither a variable nor an integer (0 or 1) nor an FD boolean functor nor reified constraint	`type_error(fd_bool_evaluable, E)`
an expression is too complex	`resource_error(too_big_fd_constraint)`
a sub-expression is an invalid reified constraint	an arithmetic constraint error (section 8.6.1, page 170)

8.7.2 `(#\)/1` - constraint NOT, `(#<=>)/2` - constraint equivalent, `(#\<=>)/2` - constraint different, `(##)/2` - constraint XOR, `(#==>)/2` - constraint imply, `(#\==>)/2` - constraint not imply, `(#/\)/2` - constraint AND, `(#\/\)/2` - constraint NAND, `(#\/)/2` - constraint OR, `(#\\/)/2` - constraint NOR

Templates

```
#\(?fd_bool_evaluable)
#<=>(?fd_bool_evaluable, ?fd_bool_evaluable)
#\<=>(?fd_bool_evaluable, ?fd_bool_evaluable)
##(?fd_bool_evaluable, ?fd_bool_evaluable)
#==>(?fd_bool_evaluable, ?fd_bool_evaluable)
#\==>(?fd_bool_evaluable, ?fd_bool_evaluable)
#/\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\/\(?fd_bool_evaluable, ?fd_bool_evaluable)
#\/(?fd_bool_evaluable, ?fd_bool_evaluable)
#\\/(?fd_bool_evaluable, ?fd_bool_evaluable)
```

Description

`#\= FdBoolExpr1` constraints `FdBoolExpr1` to be false.

`FdBoolExpr1 #<=> FdBoolExpr2` constrains `FdBoolExpr1` to be equivalent to `FdBoolExpr2`.

`FdBoolExpr1 #\<=> FdBoolExpr2` constrains `FdBoolExpr1` to be equivalent to not `FdBoolExpr2`.

`FdBoolExpr1 ## FdBoolExpr2` constrains `FdBoolExpr1` XOR `FdBoolExpr2` to be true

`FdBoolExpr1 #==> FdBoolExpr2` constrains `FdBoolExpr1` to imply `FdBoolExpr2`.

`FdBoolExpr1 #\==> FdBoolExpr2` constrains `FdBoolExpr1` to not imply `FdBoolExpr2`.

`FdBoolExpr1 #/\ FdBoolExpr2` constrains `FdBoolExpr1` AND `FdBoolExpr2` to be true.

`FdBoolExpr1 #\/\ FdBoolExpr2` constrains `FdBoolExpr1` AND `FdBoolExpr2` to be false.

`FdBoolExpr1 #\/ FdBoolExpr2` constrains `FdBoolExpr1` OR `FdBoolExpr2` to be true.

`FdBoolExpr1 #\\/ FdBoolExpr2` constrains `FdBoolExpr1` OR `FdBoolExpr2` to be false.

`FdBoolExpr1` and `FdBoolExpr2` are boolean FD expressions (section 8.7.1, page 173).

Note that `#\<=>` (not equivalent) and `##` (exclusive or) are synonymous.

These predicates post boolean constraints that are managed by the FD solver using a partial arc-consistency algorithm to reduce the domain of involved variables. The (dis)entailment of reified constraints is detected using either the bounds (for partial AC arithmetic constraints) or the full domain (for full AC arithmetic constraints).

`#<=>`, `#\<=>`, `##`, `#==>`, `#\==>`, `#/\`, `#\/\`, `#\/` and `#\\/` are predefined infix operators. `#\` is a predefined prefix operator (section 7.14.10, page 100).

Errors

Refer to the syntax of boolean FD expressions for possible errors (section 8.7.1, page 173).

Portability

GNU Prolog predicates.

8.7.3 fd_cardinality/2, fd_cardinality/3, fd_at_least_one/1, fd_at_most_one/1, fd_only_one/1

Templates

```
fd_cardinality(+fd_bool_evaluable_list, ?fd_variable)
fd_cardinality(+integer, ?fd_variable, +integer)
fd_at_least_one(+fd_bool_evaluable_list)
fd_at_most_one(+fd_bool_evaluable_list)
fd_only_one(+fd_bool_evaluable_list)
```

Description

`fd_cardinality(List, Count)` unifies `Count` with the number of constraints that are true in `List`. This is equivalent to post the constraint $\texttt{B}_1$ `+` $\texttt{B}_2$ `+ ...+` $\texttt{B}_n$ `#= Count` where each variable `Bi` is a new variable defined by the constraint $\texttt{B}_i$ `#<=>` $\texttt{C}_i$ where $\texttt{C}_i$ is the `i`*th* constraint of `List`. Each $\texttt{C}_i$ must be a boolean FD expression (section 8.7.1, page 173).

`fd_cardinality(Lower, List, Upper)` is equivalent to `fd_cardinality(List, Count), Lower #=< Count, Count #=< Upper`

`fd_at_least_one(List)` is equivalent to `fd_cardinality(List, Count), Count #>= 1`.

`fd_at_most_one(List)` is equivalent to `fd_cardinality(List, Count), Count #=< 1`.

`fd_only_one(List)` is equivalent to `fd_cardinality(List, 1)`.

Errors

`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
`Count` is neither an FD variable nor an integer	`type_error(fd_variable, Count)`
`Lower` is a variable	`instantiation_error`
`Lower` is neither a variable nor an integer	`type_error(integer, Lower)`
`Upper` is a variable	`instantiation_error`
`Upper` is neither a variable nor an integer	`type_error(integer, Upper)`
an element `E` of the `List` list is an invalid boolean expression	an FD boolean constraint (section 8.7.1, page 173)

Portability

GNU Prolog predicates.

8.8 Symbolic constraints

8.8.1 `fd_all_different/1`

Templates

`fd_all_different(+fd_variable_list)`

Description

`fd_all_different(List)` constrains all variables in `List` to take distinct values. This is equivalent to posting an inequality constraint for each pair of variables. This constraint is triggered when a variable becomes ground, removing its value from the domain of the other variables.

Errors

`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
an element `E` of the `List` list is neither a variable nor an integer nor an FD variable	`type_error(fd_variable, E)`

Portability

GNU Prolog predicate.

8.8.2 `fd_element/3`

Templates

`fd_element(?fd_variable, +integer_list, ?fd_variable)`

Description

`fd_element(I, List, X)` constraints `X` to be equal to the `I`*th* integer (from 1) of `List`.

Errors

`I` is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, I)`
`X` is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, X)`
`List` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
an element `E` of the `List` list is neither a variable nor an integer	`type_error(integer, E)`

Portability

GNU Prolog predicate.

8.8.3 `fd_element_var/3`

Templates

```
fd_element_var(?fd_variable, +fd_variable_list, ?fd_variable)
```

Description

`fd_element_var(I, List, X)` constraints `X` to be equal to the `I`*th* variable (from 1) of `List`. This constraint is similar to `fd_element/3` (section 8.8.2, page 176) but `List` can also contain FD variables (rather than just integers).

Errors

`I` is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, I)`
`X` is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, X)`
`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
an element `E` of the `List` list is neither a variable nor an integer nor an FD variable	`type_error(fd_variable, E)`

Portability

GNU Prolog predicate.

8.8.4 `fd_atmost/3`, `fd_atleast/3`, `fd_exactly/3`

Templates

```
fd_atmost(+integer, +fd_variable_list, +integer)
fd_atleast(+integer, +fd_variable_list, +integer)
fd_exactly(+integer, +fd_variable_list, +integer)
```

Description

`fd_atmost(N, List, V)` posts the constraint that at most `N` variables of `List` are equal to the value `V`.

`fd_atleast(N, List, V)` posts the constraint that at least `N` variables of `List` are equal to the value `V`.

`fd_exactly(N, List, V)` posts the constraint that at exactly `N` variables of `List` are equal to the value `V`.

These constraints are special cases of `fd_cardinality/2` (section 8.7.3, page 175) but their implementation is more efficient.

Errors

`N` is a variable	`instantiation_error`
`N` is neither a variable nor an integer	`type_error(integer, N)`
`V` is a variable	`instantiation_error`
`V` is neither a variable nor an integer	`type_error(integer, V)`
`List` is a partial list	`instantiation_error`
`List` is neither a partial list nor a list	`type_error(list, List)`
an element `E` of the `List` list is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, E)`

Portability

GNU Prolog predicates.

8.8.5 fd_relation/2, fd_relationc/2

Templates

```
fd_relation(+integer_list_list, ?fd_variable_list)
fd_relationc(+integer_list_list, ?fd_variable_list)
```

Description

`fd_relation(Relation, Vars)` constraints the tuple of variables `Vars` to be equal to one tuple of the list `Relation`. A tuple is represented by a list.

Example: definition of the boolean AND relation so that X AND Y $\Leftrightarrow$ Z:

```
and(X,Y,Z):-
        fd_relation([[0,0,0],[0,1,0],[1,0,0],[1,1,1]], [X,Y,Z]).
```

`fd_relationc(Columns, Vars)` is similar to `fd_relation/2` except that the relation is not given as the list of tuples but as the list of the columns of the relation. A column is represented by a list.

Example:

```
and(X,Y,Z):-
        fd_relationc([[0,0,1,1],[0,1,0,1],[0,0,0,1]], [X,Y,Z]).
```

Errors

`Relation` is a partial list or a list with a sub-term `E` which is a variable	`instantiation_error`
`Relation` is neither a partial list nor a list	`type_error(list, Relation)`
an element `E` of the `Relation` list is neither a variable nor an integer	`type_error(integer, E)`
`Vars` is a partial list	`instantiation_error`
`Vars` is neither a partial list nor a list	`type_error(list, Vars)`
an element `E` of the `Vars` list is neither a variable nor an integer nor an FD variable	`type_error(fd_variable, E)`

Portability

GNU Prolog predicates.

8.9 Labeling constraints

8.9.1 fd_labeling/2, fd_labeling/1, fd_labelingff/1

Templates

```
fd_labeling(+fd_variable_list, +fd_labeling_option_list)
fd_labeling(+fd_variable, +fd_labeling_option_list)
fd_labeling(+fd_variable_list)
fd_labeling(+fd_variable)
fd_labelingff(+fd_variable_list)
fd_labelingff(+fd_variable)
```

Description

`fd_labeling(Vars, Options)` assigns a value to each variable `X` of the list `Vars` according to the list of labeling options given by `Options`. `Vars` can be also a single FD variable. This predicate is re-executable on backtracking.

FD labeling options: `Options` is a list of labeling options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `variable_method(V)`: specifies the heuristics to select the variable to enumerate:
  - `standard`: no heuristics, the leftmost variable is selected.
  - `first_fail` (or `ff`): selects the variable with the smallest number of elements in its domain. If several variables have the same number of elements the leftmost variable is selected.
  - `most_constrained`: like `first_fail` but when several variables have the same number of elements selects the variable that appears in most constraints.
  - `smallest`: selects the variable that has the smallest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - `largest`: selects the variable that has the greatest value in its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - `max_regret`: selects the variable that has the greatest difference between the smallest value and the next value of its domain. If there is more than one such variable selects the variable that appears in most constraints.
  - `random`: selects randomly a variable. Each variable is only chosen once.

  The default value is `standard`.
- `reorder(true/false)`: specifies if the variable heuristics should dynamically reorder the list of variable (`true`) or not (`false`). Dynamic reordering is generally more efficient but in some cases a static ordering is faster. The default value is `true`.
- `value_method(V)`: specifies the heuristics to select the value to assign to the chosen variable:
  - `min`: enumerates the values from the smallest to the greatest (default).
  - `max`: enumerates the values from the greatest to the smallest.
  - `middle`: enumerates the values from the middle to the bounds.
  - `bounds`: enumerates the values from the bounds to the middle.
  - `random`: enumerates the values randomly. Each value is only tried once.

  The default value is `min`.

- `backtracks(B)`: unifies `B` with the number of backtracks during the enumeration.

`fd_labeling(Vars)` is equivalent to `fd_labeling(Vars, [])`.

`fd_labelingff(Vars)` is equivalent to `fd_labeling(Vars, [variable_method(ff)])`.

Errors

`Vars` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Vars` is neither a partial list nor a list	`type_error(list, Vars)`
an element `E` of the `Vars` list is neither a variable nor an integer nor an FD variable	`type_error(fd_variable, E)`
`Options` is a partial list or a list with an element `E` which is a variable	`instantiation_error`
`Options` is neither a partial list nor a list	`type_error(list, Options)`
an element `E` of the `Options` list is neither a variable nor a labeling option	`domain_error(fd_labeling_option, E)`

Portability

GNU Prolog predicates.

8.10 Optimization constraints

8.10.1 `fd_minimize/2`, `fd_maximize/2`

Templates

```
fd_minimize(+callable_term, ?fd_variable)
fd_maximize(+callable_term, ?fd_variable)
```

Description

`fd_minimize(Goal, X)` repeatedly calls `Goal` to find a value that minimizes the variable `X`. `Goal` is a Prolog goal that should instantiate `X`, a common case being the use of `fd_labeling/2` (section 8.9.1, page 179). This predicate uses a branch-and-bound algorithm with restart: each time `call(Goal)` succeeds the computation restarts with a new constraint `X #< V` where `V` is the value of `X` at the end of the last call of `Goal`. When a failure occurs (either because there are no remaining choice-points for `Goal` or because the added constraint is inconsistent with the rest of the store) the last solution is recomputed since it is optimal.

`fd_maximize(Goal, X)` is similar to `fd_minimize/2` but `X` is maximized.

Errors

`Goal` is a variable	`instantiation_error`
`Goal` is neither a variable nor a callable term	`type_error(callable, Goal)`
The predicate indicator `Pred` of `Goal` does not correspond to an existing procedure and the value of the `unknown` Prolog flag is `error` (section 7.22.1, page 133)	`existence_error(procedure, Pred)`
`X` is neither a variable nor an FD variable nor an integer	`type_error(fd_variable, X)`

Portability

GNU Prolog predicates.

9 Interfacing Prolog and C

9.1 Calling C from Prolog

9.1.1 Introduction

This interface allows a Prolog predicate to call a C function. Here are some features of this facility:

- implicit Prolog $\leftrightarrow$ C data conversions for simple types.
- functions to handle complex types.
- error detection depending on the type of the argument.
- different kinds of arguments: input, output or input/output.
- possibility to write non-deterministic code.

This interface can then be used to write both simple and complex C routines. A simple routine uses either input or output arguments which type is simple. In that case the user does not need any knowledge of Prolog data structures since all Prolog $\leftrightarrow$ C data conversions are implicitly achieved. To manipulate complex terms (lists, structures) a set of functions is provided. Finally it is also possible to write non-deterministic C code.

9.1.2 `foreign/2` directive

`foreign/2` directive (section 6.1.14, page 45) declares a C function interface. The general form is `foreign(Template, Options)` which defines an interface predicate whose prototype is `Template` according to the options given by `Options`. `Template` is a callable term specifying the type/mode of each argument of the associated Prolog predicate.

Foreign options: `Options` is a list of foreign options. If this list contains contradictory options, the rightmost option is the one which applies. Possible options are:

- `fct_name(F)`: `F` is an atom representing the name of the C function to call. By default the name of the C function is the same as the principal functor of `Template`. In any case, the atom associated to the name of the function must conforms to the syntax of C identifiers.
- `return(boolean/none/jump)`: specifies the value returned by the C function:
  - `boolean`: the type of the function is `Bool` (returns `TRUE` on success, `FALSE` otherwise).
  - `none`: the type of the function is `void` (no returned value).
  - `jump`: the type of the function is `void(*)()` (returns the address of a Prolog code to execute).

  The default value is `boolean`.
- `bip_name(Name, Arity)`: initializes the error context with `Name` and `Arity`. If an error occurs this information is used to indicate from which predicate the error occurred (section 5.3.1, page 37). It is also possible to prevent the initialization of the error context using `bip_name(none)`. By default `Name` and `Arity` are set to the functor and arity of `Template`.
- `choice_size(N)`: this option specifies that the function implements a non-deterministic code. `N` is an integer specifying the size needed by the non-deterministic C function. This facility is explained later (section 9.1.7, page 186). By default a foreign function is deterministic.

`foreign(Template)` is equivalent to `foreign(Template, [])`.

Foreign modes and types: each argument of `Template` specifies the foreign mode and type of the corresponding argument. This information is used to check the type of effective arguments at run-time and to perform Prolog $\leftrightarrow$ C data conversions. Each argument of `Template` is formed with a mode symbol followed by a type name. Possible foreign modes are:

- `+`: input argument.
- `-`: output argument.
- `?`: input/output argument.

Possible foreign types are:

Foreign type	Prolog type	C type	Description of the C type
`integer`	integer	`long`	value of the integer
`positive`	positive integer	`long`	value of the integer
`float`	floating point number	`double`	value of the floating point number
`number`	number	`double`	value of the number
`atom`	atom	`int`	internal key of the atom
`boolean`	boolean	`int`	value of the boolean (0=`false`, 1=`true`)
`char`	character	`int`	value of (the code of) the character
`code`	character code	`int`	value of the character-code
`byte`	byte	`int`	value of the byte
`in_char`	in-character	`int`	value of the character or `-1` for end-of-file
`in_code`	in-character code	`int`	value of the character-code or `-1` for end-of-file
`in_byte`	in-byte	`int`	value of the byte or `-1` for the end-of-file
`string`	atom	`char *`	C string containing the name of the atom
`chars`	character list	`char *`	C string containing the characters of the list
`codes`	character-code list	`char *`	C string containing the characters of the list
`term`	Prolog term	`PlTerm`	generic Prolog term

Simple foreign type: a simple type is any foreign type listed in the above tabled except `term`. A simple foreign type is an atomic term (character and character-code lists are in fact lists of constants). Each simple foreign type is converted to/from a C type to simplify the writing of the C function.

Complex foreign type: type foreign type `term` refers to any Prolog term (e.g. lists, structures...). When such an type is specified the argument is passed to the C function as a `PlTerm` (GNU Prolog C type equivalent to a `long`). Several functions are provided to manipulate `PlTerm` variables (section 9.2, page 190). Since the original term is passed to the function it is possible to read its value or to unify it. So the meaning of the mode symbol is less significant. For this reason it is possible to omit the mode symbol. In that case `term` is equivalent to `+term`.

9.1.3 The C function

The C code is written in a C file which must first include the GNU Prolog header file called `gprolog.h`. This file contains all GNU Prolog C definitions (constants, types, prototypes,...).

The type returned by a C function depends on the value of the `return` foreign option (section 9.1.2, page 183). If it is `boolean` then the C function is of type `Bool` and shall return `TRUE` in case of success and `FALSE` otherwise. If the `return` option is `none` the C function is of type `void`. Finally if it is `jump`, the function shall return the address of a Prolog predicate and, at the exit of the function, the control is given to that predicate.

The type of the arguments of the C function depends on the mode and type declaration specified in `Template` for the corresponding argument as explained in the following sections.

9.1.4 Input arguments

An input argument is tested at run-time to check if its type conforms to the foreign type and then it is passed to the C function. The type of the associated C argument is given by the above table (section 9.1.2, page 183). For instance, the effective argument `Arg` associated to `+positive` foreign declaration is submitted to the following process:

- if `Arg` is a variable an `instantiation_error` is raised.
- if `Arg` is neither a variable nor an integer a `type_error(integer, Arg)` is raised.
- if `Arg` is an integer < 0 a `domain_error(not_less_than_zero, Arg)` is raised.
- otherwise the value of `Arg` is passed to the C is passed to the C function as an integer (`long`).

When `+string` is specified the string passed to the function is the internal string of the corresponding atom and should not be modified.

When `+term` is specified the term passed to the function is the original Prolog term. It can be read and/or unified. It is also the case when `term` is specified without any mode symbol.

9.1.5 Output arguments

An output argument is tested at run-time to check if its type conforms to the foreign type and it is unified with the value set by the C function. The type of the associated C argument is a pointer to the type given by the above table (section 9.1.2, page 183). For instance, the effective argument `Arg` associated to `-positive` foreign declaration is handled as follows:

- if `Arg` is neither a variable nor an integer a `type_error(integer, Arg)` is raised.
- if `Arg` is an integer < 0 a `domain_error(not_less_than_zero, Arg)` is raised.
- otherwise a pointer to an integer (`long *`) is passed to the C function. If the function returns `TRUE` the integer stored at this location is unified with `Arg`.

When `-term` is specified, the function must construct a term into the its corresponding argument (which is of type `PlTerm *`). At the exit of the function this term will be unified with the actual predicate argument.

9.1.6 Input/output arguments

Basically an input/output argument is treated as in input argument if it is not a variable, as an output argument otherwise. The type of the associated C argument is a pointer to a `FIOArg` (GNU Prolog C type) defined as follows:

```
typedef struct
    {
     Bool is_var;
     Bool unify;
     union
        {
         long   l;
         char  *s;
         double d;
        }value;
    }FIOArg;
```

The field `is_var` is set to `TRUE` if the argument is a variable and `FALSE` otherwise. This value can be tested by the C function to determine which treatment to perform. The field `unify` controls whether the effective argument must be unified at the exit of the C function. Initially `unify` is set to the same value as `is_var` (i.e. a variable argument will be unified while a non-variable argument will not) but it can be modified by the C function. The field `value` stores the value of the argument. It is declared as a C `union` since there are several kinds of value types. The field `s` is used for C strings, `d` for C doubles and `l` otherwise (`int`, `long`, `PlTerm`). if `is_var` is `FALSE` then `value` contains the input value of the argument with the same conventions as for input arguments (section 9.1.4, page 185). At the exit of the function, if unify is `TRUE` `value` must contain the value to unify with the same conventions as for output arguments (section 9.1.5, page 185).

For instance, the effective argument `Arg` associated to `?positive` foreign declaration is handled as follows:

- if `Arg` is a variable `is_var` and `unify` are set to `TRUE` else to `FALSE` and its value is copied in `value.l`.
- if `Arg` is neither a variable nor an integer a `type_error(integer, Arg)` is raised.
- if `Arg` is an integer < 0 a `domain_error(not_less_than_zero, Arg)` is raised.
- otherwise a pointer to the `FIOArg` (`FIOArg *`) is passed to the C function. If the function returns `TRUE` and if `unify` is TRUE the value stored in `value.l` is unified with `Arg`.

9.1.7 Writing non-deterministic C code

The interface allows the user to write non-deterministic C code. When a C function is non-deterministic, a choice-point is created for this function. When a failure occurs, if all more recent non-deterministic code are finished, the function is re-invoked. It is then important to inform Prolog when there is no more solution (i.e. no more choice) for a non-deterministic code. So, when no more choices remains the function must remove the choice-point. The interface increments a counter each time the function is re-invoked. At the first call this counter is equal to 0. This information allows the function to detect its first call. When writing non-deterministic code, it is often useful to record data between consecutive re-invocations of the function. The interface maintains a buffer to record such an information. The size of this buffer is given by `choice_size(N)` when using `foreign/2` (section 9.1.2, page 183). This size is the number of (consecutive) `long`*s* needed by the C function. Inside the function it is possible to call the following functions/macros:

```
void Get_Choice_Counter(void)
TYPE Get_Choice_Buffer (TYPE)
void No_More_Choice    (void)
```

The function `Get_Choice_Counter()` returns the value of the invocation counter (0 at the first call).

The macro `Get_Choice_Buffer(`*`TYPE`*`)` returns a pointer to the buffer (casted to *`TYPE`*).

The function `No_More_Choice()` deletes the choice point associated to the function.

9.1.8 Example: input and output arguments

All examples presented here can be found in the `ExamplesC` sub-directory of the distribution, in the files `examp.pl` (Prolog part) and `examp_c.c` (C part).

Let us define a predicate `first_occurrence(A, C, P)` which unifies `P` with the position (from 0) of the first occurrence of the character `C` in the atom `A`. The predicate must fail if `C` does not appear in `A`.

In the prolog file `examp.pl`:

```
:- foreign(first_occurrence(+string, +char, -positive)).
```

In the C file examp_c.c:

```
#include <string.h>
#include "gprolog.h"

Bool
first_occurrence(char *str, long c, long *pos)
{
  char *p;

  p = strchr(str, c);
  if (p == NULL)                /* C does not appear in A */
    return FALSE;               /* fail */

  *pos = p - str;               /* set the output argument */
  return TRUE;                  /* succeed */
}
```

The compilation produces an executable called examp:

```
% gplc examp.pl examp_c.c
```

Examples of use:

```
| ?- first_occurrence(prolog, p, X).

X = 0

| ?- first_occurrence(prolog, k, X).

no

| ?- first_occurrence(prolog, A, X).
{exception: error(instantiation_error,first_occurrence/3)}

| ?- first_occurrence(prolog, 1 ,X).
{exception: error(type_error(character,1),first_occurrence/3)}
```

9.1.9 Example: non-deterministic code

We here define a predicate `occurrence(A, C, P)` which unifies `P` with the position (from 0) of one occurrence of the character `C` in the atom `A`. The predicate will fail if `C` does not appear in `A`. The predicate is re-executable on backtracking. The information that must be recorded between two invocations of the function is the next starting position in `A` to search for `C`.

In the prolog file examp.pl:

```
:- foreign(occurrence(+string, +char, -positive), [choice_size(1)]).
```

In the C file examp_c.c:

```
#include <string.h>
#include "gprolog.h"

Bool
occurrence(char *str, long c, long *pos)
{
```

```
  char **info_pos;
  char *p;

  info_pos = Get_Choice_Buffer(char **); /* recover the buffer */

  if (Get_Choice_Counter() == 0)        /* first invocation ? */
    *info_pos = str;

  p = strchr(*info_pos, c);
  if (p == NULL)                  /* C does not appear */
    {
      No_More_Choice();           /* remove choice-point */
      return FALSE;               /* fail */
    }

  *pos = p - str;                 /* set the output argument */
  *info_pos = p + 1;              /* update next starting pos */
  return TRUE;                    /* succeed */
}
```

The compilation produces an executable called `examp`:

```
% gplc examp.pl examp_c.c
```

Examples of use:

```
| ?- occurrence(prolog, o, X).

X = 2 ?                   (here the user presses ; to compute another solution)

X = 4 ?                   (here the user presses ; to compute another solution)

no                        (no more solution)

| ?- occurrence(prolog, k, X).

no
```

In the first example when the second (the last) occurrence is found (`X=4`) the choice-point remains and the failure is detected only when another solution is requested (by pressing `;`). It is possible to improve this behavior by deleting the choice-point when there is no more occurrence. To do this it is necessary to do one search ahead. The information stored is the position of the next occurrence. Let us define such a behavior for the predicate `occurrence2/3`.

In the prolog file `examp.pl`:

```
:- foreign(occurrence2(+string, +char, -positive), [choice_size(1)]).
```

In the C file `examp_c.c`:

```
#include <string.h>
#include "gprolog.h"

Bool
occurrence2(char *str, long c, long *pos)
{
  char **info_pos;
  char *p;

  info_pos = Get_Choice_Buffer(char **); /* recover the buffer */
```

```
  if (Get_Choice_Counter() == 0) /* first invocation ? */
    {
      p = strchr(str, c);
      if (p == NULL)            /* C does not appear at all */
        {
          No_More_Choice();     /* remove choice-point */
          return FALSE;         /* fail */
        }

      *info_pos = p;
    }
                                /* info_pos = an occurrence */
  *pos = *info_pos - str;       /* set the output argument */

  p = strchr(*info_pos + 1, c);
  if (p == NULL)                /* no more occurrence */
    No_More_Choice();           /* remove choice-point */
  else
    *info_pos = p;              /* else update next solution */

  return TRUE;                  /* succeed */
}
```

Examples of use:

```
| ?- occurrence2(prolog, l, X).
```

`X = 3` (here the user is not prompted since there is no more alternative)

```
| ?- occurrence2(prolog, o, X).
```

`X = 2 ?` (here the user presses `;` to compute another solution)

`X = 4` (here the user is not prompted since there is no more alternative)

9.1.10 Example: input/output arguments

We here define a predicate `char_ascii(Char, Code)` which converts in both directions the character `Char` and its character-code `Code`. This predicate is then similar to `char_code/2` (section 7.19.4, page 116).

In the prolog file `examp.pl`:

```
:- foreign(char_ascii(?char, ?code), [fct_name('Char_Ascii')]).
```

In the C file `examp_c.c`:

```
#include "gprolog.h"

Bool
char_ascii(FIOArg *c, FIOArg *ascii)
{
  if (!c->is_var)               /* Char is not a variable */
    {
      ascii->unify = TRUE;      /* enforce unif. of Code */
      ascii->value.l = c->value.l; /* set Code */
      return TRUE;              /* succeed */
```

```
    }

    if (ascii->is_var)            /* Code is also a variable */
      Pl_Err_Instantiation();     /* emit instantiation_error */

    c->value.l = ascii->value.l;  /* set Char */
    return TRUE;                  /* succeed */
  }
```

If `Char` is instantiated it is necessary to enforce the unification of `Code` since it could be instantiated. Recall that by default if an input/output argument is instantiated it will not be unified at the exit of the function (section 9.1.6, page 185). If both `Char` and `Code` are variables the function raises an `instantiation_error`. The way to raise Prolog errors is described later (section 9.3, page 196).

The compilation produces an executable called `examp`:

```
% gplc examp.pl examp_c.c
```

Examples of use:

```
| ?- char_ascii(a, X).

X = 97

| ?- char_ascii(X, 65).

X = 'A'

| ?- char_ascii(a, 12).

no

| ?- char_ascii(X, X).
{exception: error(instantiation_error,char_ascii/2)}

| ?- char_ascii(1, 12).
{exception: error(type_error(character,1),char_ascii/2)}
```

9.2 Manipulating Prolog terms

9.2.1 Introduction

In the following we presents a set of functions to manipulate Prolog terms. For simple foreign terms the functions manipulate simple C types (section 9.1.2, page 183).

Functions managing lists handle an array of 2 elements (of type `PlTerm`) containing the terms corresponding to the head and the tail of the list. For the empty list `NULL` is passed as the array. These functions require to flatten a list in each sub-list. To simplify the management of proper lists (i.e. lists terminated by `[]`) a set of functions is provided that handle the number of elements of the list (an integer) and an array whose elements (of type `PlTerm`) are the elements of the list. The caller of these functions must provide the array.

Functions managing compound terms handle a functor (the principal functor of the term), an arity $N \geq 0$ and an array of N elements (of type `PlTerm`) containing the sub-terms of the compound term. Since a list is a special case of compound term (functor = `'.'` and arity=2) it is possible to use any function managing compound terms to deal with a list but the error detection is not the same. Indeed many functions check if the Prolog argument is correct. The name of a read or unify function checking the Prolog arguments is of the form `Name_Check()`. For each of these functions there is a also check-free version called `Name()`. We then only present the name of checking functions.

9.2.2 Managing Prolog atoms

Each atom has a unique internal key which corresponds to its index in the GNU Prolog atom table. It is possible to obtain the information about an atom and to create new atoms using:

```
char *Atom_Name           (int atom)
int   Atom_Length         (int atom)
Bool  Atom_Needs_Quote    (int atom)
Bool  Atom_Needs_Scan     (int atom)
Bool  Is_Valid_Atom       (int atom)
int   Create_Atom         (char *str)
int   Create_Allocate_Atom(char *str)
int   Find_Atom           (char *str)
int   ATOM_CHAR           (char c)
int   atom_nil
int   atom_false
int   atom_true
int   atom_end_of_file
```

The macro `Atom_Name(atom)` returns the internal string of `atom` (this string should not be modified). The function `Atom_Lengh(atom)` returns the length (of the name) of `atom`.

The function `Atom_Needs_Scan(atom)` indicates if the canonical form of `atom` needs to be quoted as done by `writeq/2` (section 7.14.6, page 96). In that case `Atom_Needs_Scan(atom)` indicates if this simply comes down to write quotes around the name of `atom` or if it necessary to scan each character of the name because there are some non-printable characters (or included quote characters). The function `Is_Valid_Atom(atom)` is true only if `atom` is the internal key of an existing atom.

The function `Create_Atom(str)` adds a new atom whose name is the content of `str` to the system and returns its internal key. If the atom already exists its key is simply returned. The string `str` passed to the function should not be modified later. The function `Create_Allocate_Atom(str)` is provided when this condition cannot be ensured. It simply makes a dynamic copy of `str`.

The function `Find_Atom(str)` returns the internal key of the atom whose name is `str` or `-1` if does not exist.

All atoms corresponding to a single character already exist and their key can be obtained via the macro `ATOM_CHAR`. For instance `ATOM_CHAR('.')` is the atom associated to `'.'` (this atom is the functor of lists). The other variables correspond to the internal key of frequently used atoms: `[]`, `false`, `true` and `end_of_file`.

9.2.3 Reading Prolog terms

The name of all functions presented here are of the form `Rd_Name_Check()`. They all check the validity of the Prolog term to read emitting appropriate errors if necessary. Each function has a check-free version called `Rd_Name()`.

Simple foreign types: for each simple foreign type (section 9.1.2, page 183) there is a read function (used by the interface when an input argument is provided):

```
long    Rd_Integer_Check  (PlTerm term)
```

```
long    Rd_Positive_Check (PlTerm term)
double  Rd_Float_Check    (PlTerm term)
double  Rd_Number_Check   (PlTerm term)
int     Rd_Atom_Check     (PlTerm term)
int     Rd_Boolean_Check  (PlTerm term)
int     Rd_Char_Check     (PlTerm term)
int     Rd_In_Char_Check  (PlTerm term)
int     Rd_Code_Check     (PlTerm term)
int     Rd_In_Code_Check  (PlTerm term)
int     Rd_Byte_Check     (PlTerm term)
int     Rd_In_Byte_Check  (PlTerm term)
char   *Rd_String_Check   (PlTerm term)
char   *Rd_Chars_Check    (PlTerm term)
char   *Rd_Codes_Check    (PlTerm term)
int     Rd_Chars_Str_Check(PlTerm term, char *str)
int     Rd_Codes_Str_Check(PlTerm term, char *str)
```

All functions returning a C string (`char *`) use a same buffer. The function `Rd_Chars_Str_Check()` is similar to `Rd_Chars_Check()` but accepts as argument a string to store the result and returns the length of that string (which is also the length of the Prolog list). Similarly for `Rd_Codes_Str_Check()`.

Complex terms: the following functions return the sub-arguments (terms) of complex terms as an array of `PlTerm` except `Rd_Proper_List_Check()` which returns the size of the list read (and initializes the array `element`). Refer to the introduction of this section for more information about the arguments of complex functions (section 9.2.1, page 190).

```
int     Rd_Proper_List_Check(PlTerm term, PlTerm *arg)
PlTerm *Rd_List_Check       (PlTerm term)
PlTerm *Rd_Compound_Check   (PlTerm term, int *functor, int *arity)
PlTerm *Rd_Callable_Check   (PlTerm term, int *functor, int *arity)
```

9.2.4 Unifying Prolog terms

The name of all functions presented here are of the form `Un_`*`Name`*`_Check()`. They all check the validity of the Prolog term to unify emitting appropriate errors if necessary. Each function has a check-free version called `Un_`*`Name`*`()`.

Simple foreign types: for each simple foreign type (section 9.1.2, page 183) there is an unify function (used by the interface when an output argument is provided):

```
Bool Un_Integer_Check (long n,    PlTerm term)
Bool Un_Positive_Check(long n,    PlTerm term)
Bool Un_Float_Check   (double n,  PlTerm term)
Bool Un_Number_Check  (double n,  PlTerm term)
Bool Un_Atom_Check    (int atom,  PlTerm term)
Bool Un_Boolean_Check (int b,     PlTerm term)
Bool Un_Char_Check    (int c,     PlTerm term)
Bool Un_In_Char_Check (int c,     PlTerm term)
Bool Un_Code_Check    (int c,     PlTerm term)
Bool Un_In_Code_Check (int c,     PlTerm term)
Bool Un_Byte_Check    (int b,     PlTerm term)
Bool Un_In_Byte_Check (int b,     PlTerm term)
Bool Un_String_Check  (char *str, PlTerm term)
Bool Un_Chars_Check   (char *str, PlTerm term)
Bool Un_Codes_Check   (char *str, PlTerm term)
```

The function `Un_Number_Check(n, term)` unifies `term` with an integer if `n` is an integer, with a floating point number otherwise. The function `Un_String_Check(str, term)` creates the atom corresponding to `str` and then unifies term with it (same as `Un_Atom_Check(Create_Allocate_Atom(str), term)`).

Complex terms: the following functions accept the sub-arguments (terms) of complex terms as an array of `PlTerm`. Refer to the introduction of this section for more information about the arguments of complex functions (section 9.2.1, page 190).

```
Bool Un_Proper_List_Check(int size, PlTerm *arg, PlTerm term)
Bool Un_List_Check       (PlTerm *arg, PlTerm term)
Bool Un_Compound_Check   (int functor, int arity, PlTerm *arg,
                          PlTerm term)
Bool Un_Callable_Check   (int functor, int arity, PlTerm *arg,
                          PlTerm term)
```

All these functions check the type of the term to unify and return the result of the unification. Generally if an unification fails the C function returns `FALSE` to enforce a failure. However if there are several arguments to unify and if an unification fails then the C function returns `FALSE` and the type of other arguments has not been checked. Normally all error cases are tested before doing any work to be sure that the predicate fails/succeeds only if no error condition is satisfied. So a good method is to check if the validity of all arguments to unify and later to do the unification (using check-free functions). Obviously if there is only one to unify it is more efficient to use a unify function checking the argument. For the other cases the interface provides a set of functions to check the type of a term.

Simple foreign types: for each simple foreign type (section 9.1.2, page 183) there is check-for-unification function (used by the interface when an output argument is provided):

```
void Check_For_Un_Integer (PlTerm term)
void Check_For_Un_Positive(PlTerm term)
void Check_For_Un_Float   (PlTerm term)
void Check_For_Un_Number  (PlTerm term)
void Check_For_Un_Atom    (PlTerm term)
void Check_For_Un_Boolean (PlTerm term)
void Check_For_Un_Char    (PlTerm term)
void Check_For_Un_In_Char (PlTerm term)
void Check_For_Un_Code    (PlTerm term)
void Check_For_Un_In_Code (PlTerm term)
void Check_For_Un_Byte    (PlTerm term)
void Check_For_Un_In_Byte (PlTerm term)
void Check_For_Un_String  (PlTerm term)
void Check_For_Un_Chars   (PlTerm term)
void Check_For_Un_Codes   (PlTerm term)
```

Complex terms: the following functions check the validity of complex terms:

```
void Check_For_Un_List    (PlTerm term)
void Check_For_Un_Compound(PlTerm term)
void Check_For_Un_Callable(PlTerm term)
void Check_For_Un_Variable(PlTerm term)
```

The function `Check_For_Un_List(term)` checks if `term` can be unified with a list. This test is done for the entire list (not only for the functor/arity of `term` but also recursively on the tail of the list). The function `Check_For_Un_Variable(term)` ensures that `term` is not currently instantiated. These functions can be defined using functions to test the type of a Prolog term (section 9.2.6, page 194) and functions to raise Prolog errors (section 9.3, page 196). For instance `Check_For_Un_List(term)` is defined as follows:

```
void Check_For_Un_List(PlTerm term)
{
 if (!Blt_List_Or_Partial_List(term))
     Pl_Err_Type(type_list, term);
}
```

9.2.5 Creating Prolog terms

These functions are provided to creates Prolog terms. Each function returns a `PlTerm` containing the created term.

Simple foreign types: for each simple foreign type (section 9.1.2, page 183) there is a creation function:

```
PlTerm Mk_Integer (long n)
PlTerm Mk_Positive(long n)
PlTerm Mk_Float   (double n)
PlTerm Mk_Number  (double n)
PlTerm Mk_Atom    (int atom)
PlTerm Mk_Boolean (int b)
PlTerm Mk_Char    (int c)
PlTerm Mk_In_Char (int c)
PlTerm Mk_Code    (int c)
PlTerm Mk_In_Code (int c)
PlTerm Mk_Byte    (int b)
PlTerm Mk_In_Byte (int b)
PlTerm Mk_String  (char *str)
PlTerm Mk_Chars   (char *str)
PlTerm Mk_Codes   (char *str)
```

The function `Mk_Number(n, term)` initializes `term` with an integer if `n` is an integer, with a floating point number otherwise. The function `Mk_String(str)` first creates an atom corresponding to `str` and then returns that Prolog atom (i.e. equivalent to `Mk_Atom(Create_Allocate_Atom(str))`).

Complex terms: the following functions accept the sub-arguments (terms) of complex terms as an array of `PlTerm`. Refer to the introduction of this section for more information about the arguments of complex functions (section 9.2.1, page 190).

```
PlTerm Mk_Proper_List(int size, PlTerm *arg)
PlTerm Mk_List       (PlTerm *arg)
PlTerm Mk_Compound   (int functor, int arity, PlTerm *arg)
PlTerm Mk_Callable   (int functor, int arity, PlTerm *arg)
```

9.2.6 Testing the type of Prolog terms

The following functions test the type of a Prolog term. Each function corresponds to a type testing built-in predicate (section 7.1.1, page 49).

```
Bool Blt_Var                (PlTerm term)
Bool Blt_Non_Var            (PlTerm term)
Bool Blt_Atom               (PlTerm term)
Bool Blt_Integer            (PlTerm term)
Bool Blt_Float              (PlTerm term)
Bool Blt_Number             (PlTerm term)
Bool Blt_Atomic             (PlTerm term)
Bool Blt_Compound           (PlTerm term)
Bool Blt_Callable           (PlTerm term)
Bool Blt_List               (PlTerm term)
Bool Blt_Partial_List       (PlTerm term)
Bool Blt_List_Or_Partial_List(PlTerm term)
Bool Blt_Fd_Var             (PlTerm term)
Bool Blt_Non_Fd_Var         (PlTerm term)
Bool Blt_Generic_Var        (PlTerm term)
```

```
Bool Blt_Non_Generic_Var     (PlTerm term)
int  Type_Of_Term            (PlTerm term)
int  List_Length             (PlTerm list)
```

The function `Type_Of_Term(term)` returns the type of `term`, the following constants can be used to test this type (e.g. in a `switch` instruction):

- `PLV`: Prolog variable.
- `FDV`: finite domain variable.
- `INT`: integer.
- `FLT`: floating point number.
- `ATM`: atom.
- `LST`: list.
- `STC`: structure

The tag `LST` means a term whose principal functor is `'.'` and whose arity is 2 (recall that the empty list is the atom `[]`). The tag `STC` means any other compound term.

The function `List_Length(list)` returns the number of elements of the `list` (`0` for the empty list). If list is not a list this function returns `-1`.

9.2.7 Comparing Prolog terms

The following functions compares Prolog terms. Each function corresponds to a comparison built-in predicate (section 7.3.2, page 51).

```
Bool Blt_Term_Eq (PlTerm term1, PlTerm term2)
Bool Blt_Term_Neq(PlTerm term1, PlTerm term2)
Bool Blt_Term_Lt (PlTerm term1, PlTerm term2)
Bool Blt_Term_Lte(PlTerm term1, PlTerm term2)
Bool Blt_Term_Gt (PlTerm term1, PlTerm term2)
Bool Blt_Term_Gte(PlTerm term1, PlTerm term2)
```

All these functions are based on a general comparison function returning a negative integer if `term1` is less than `term2`, 0 if they are equal and a positive integer otherwise:

```
int Term_Compare(PlTerm term1, PlTerm term2)
```

9.2.8 Copying Prolog terms

The following functions make a copy of a Prolog term:

```
void Copy_Term           (PlTerm *dst_adr, PlTerm *src_adr)
void Copy_Contiguous_Term(PlTerm *dst_adr, PlTerm *src_adr)
int  Term_Size           (PlTerm term)
```

The function `Copy_Term(dst_adr, src_adr)` makes a copy of the term located at `src_adr` and stores it from the address given by `dst_adr`. The result is a contiguous term. If it can be ensured that the source term is a contiguous term (i.e. result of a previous copy) the function `Copy_Contiguous_Term()` can be used instead (it is faster). In any case, sufficient space should be available for the copy (i.e. from `dst_adr`). The function `Term_Size(term)` returns the number of `PlTerm` needed by `term`.

9.2.9 Comparing and evaluating arithmetic expressions

The following functions compare arithmetic expressions. Each function corresponds to a comparison built-in predicate (section 7.6.3, page 60).

```
Bool Blt_Eq (PlTerm expr1, PlTerm expr2)
Bool Blt_Neq(PlTerm expr1, PlTerm expr2)
Bool Blt_Lt (PlTerm expr1, PlTerm expr2)
Bool Blt_Lte(PlTerm expr1, PlTerm expr2)
Bool Blt_Gt (PlTerm expr1, PlTerm expr2)
Bool Blt_Gte(PlTerm expr1, PlTerm expr2)
```

The following function evaluates the expression `expr` and stores its result as a Prolog number (integer or floating point number) in `result`:

```
void Math_Load_Value(PlTerm expr, PlTerm *result)
```

This function can be followed by a read function (section 9.2.3, page 191) to obtain the result.

9.3 Raising Prolog errors

The following functions allows a C function to raise a Prolog error. Refer to the section concerning Prolog errors for more information about the effect of raising an error (section 5.3, page 37).

9.3.1 Managing the error context

When one of the following error function is invoked it refers to the implicit error context (section 5.3.1, page 37). This context indicates the name and the arity of the concerned predicate. When using a `foreign/2` declaration this context is set by default to the name and arity of the associated Prolog predicate. This can be controlled using the `bip_name` option (section 9.1.2, page 183). In any case, the following functions can also be used to modify this context:

```
void Set_C_Bip_Name  (char *functor, int arity)
void Unset_C_Bip_Name(void)
```

The function `Set_C_Bip_Name(functor, arity)` initializes the context of the error with `functor` and `arity` (if `arity`<0 only `functor` is significant). The function `Unset_C_Bip_Name()` removes such an initialization (the context is then reset to the last `Functor/Arity` set by a call to `set_bip_name/2` (section 7.22.3, page 135). This is useful when writing a C routine to define a context for errors occurring in this routine and, before exiting to restore the previous context.

9.3.2 Instantiation error

The following function raises an instantiation error (section 5.3.2, page 37):

```
void Pl_Err_Instantiation(void)
```

9.3.3 Type error

The following function raises a type error (section 5.3.3, page 38):

```
void Pl_Err_Type(int atom_type, PlTerm culprit)
```

`atom_type` is (the internal key of) the atom associated to the expected type. For each type name *T* there is a corresponding predefined atom stored in a global variable whose name is of the form `type_`*T*. `culprit` is the argument which caused the error.

Example: `x` is an atom while an integer was expected: `Pl_Err_Type(type_integer, x)`.

9.3.4 Domain error

The following function raises a domain error (section 5.3.4, page 38):

```
void Pl_Err_Domain(int atom_domain, PlTerm culprit)
```

`atom_domain` is (the internal key of) the atom associated to the expected domain. For each domain name *D* there is a corresponding predefined atom stored in a global variable whose name is of the form `domain_`*D*. `culprit` is the argument which caused the error.

Example: `x` is < 0 but should be ≥ 0: `Pl_Err_Domain(domain_not_less_than_zero, x)`.

9.3.5 Existence error

The following function raises an existence error (section 5.3.5, page 39):

```
void Pl_Err_Existence(int atom_object, PlTerm culprit)
```

`atom_object` is (the internal key of) the atom associated to the type of the object. For each object name *O* there is a corresponding predefined atom stored in a global variable whose name is of the form `existence_`*O*. `culprit` is the argument which caused the error.

Example: `x` does not refer to an existing source: `Pl_Err_Existence(existence_source_sink, x)`.

9.3.6 Permission error

The following function raises a permission error (section 5.3.6, page 39):

```
void Pl_Err_Permission(int atom_operation, int atom_permission, PlTerm culprit)
```

`atom_operation` is (the internal key of) the atom associated to the operation which caused the error. For each operation name *O* there is a corresponding predefined atom stored in a global variable whose name is of the form `permission_operation_`*O*. `atom_permission` is (the internal key of) the atom associated to the tried permission. For each permission name *P* there is a corresponding predefined atom stored in a global variable whose name is of the form `permission_type_`*P*. `culprit` is the argument which caused the error.

Example: reading from an output stream `x`: `Pl_Err_Permission(permission_operation_input, permission_type_stream, x)`.

9.3.7 Representation error

The following function raises a representation error (section 5.3.7, page 39):

```
void Pl_Err_Representation(int atom_limit)
```

`atom_limit` is (the internal key of) the atom associated to the reached limit. For each limit name *L* there is a corresponding predefined atom stored in a global variable whose name is of the form `representation_L`.

Example: an arity too big occurs: `Pl_Err_Representation(representation_max_arity)`.

9.3.8 Evaluation error

The following function raises an evaluation error (section 5.3.8, page 40):

```
void Pl_Err_Evaluation(int atom_error)
```

`atom_error` is (the internal key of) the atom associated to the error. For each evaluation error name *E* there is a corresponding predefined atom stored in a global variable whose name is of the form `evaluation_E`.

Example: a division by zero occurs: `Pl_Err_Evaluation(evluation_zero_divisor)`.

9.3.9 Resource error

The following function raises a resource error (section 5.3.9, page 40):

```
void Pl_Err_Resource(int atom_resource)
```

`atom_resource` is (the internal key of) the atom associated to the resource. For each resource error name *R* there is a corresponding predefined atom stored in a global variable whose name is of the form `resource_R`.

Example: too many open streams: `Pl_Err_Resource(resource_too_many_open_streams)`.

9.3.10 Syntax error

The following function raises a syntax error (section 5.3.10, page 40):

```
void Pl_Err_Syntax(int atom_error)
```

`atom_error` is (the internal key of) the atom associated to the error. There is no predefined syntax error atoms.

Example: a / is expected: `Pl_Err_Syntax(Create_Atom("/ expected"))`.

The following function emits a syntax error according to the value of the `syntax_error` Prolog flag (section 7.22.1, page 133). This function can then return (if the value of the flag is either `warning` or `fail`). In that case the calling function should fail (e.g. returning `FALSE`). This function accepts a file name (the empty string C `""` can be passed), a line and column number and an error message string. Using this function makes it possible to further call the built-in predicate `syntax_error_info/4` (section 7.14.4, page 95):

```
void Emit_Syntax_Error(char *file_name, int line, int column, char *message)
```

Example: a / is expected: `Emit_Syntax_Error("data", 10, 30, "/ expected")`.

9.3.11 System error

The following function raises a system error (4.3.11, page *):

```
void Pl_Err_System(int atom_error)
```

`atom_error` is (the internal key of) the atom associated to the error. There is no predefined system error atoms.

Example: an invalid pathname is given: `Pl_Err_System(Create_Atom("invalid path name"))`.

The following function emits a system error associated to an operating system error according to the value of the `os_error` Prolog flag (section 7.22.1, page 133). This function can then return (if the value of the flag is either `warning` or `fail`). In that case the calling function should fail (e.g. returning `FALSE`). This function uses the value of the `errno` C library variable:

```
void Os_Error(void)
```

Example: a call to the C Unix function `chdir(3)` returns `-1`: `Os_Error()`.

9.4 Calling Prolog from C

9.4.1 Introduction

The following functions allows a C function to call a Prolog predicate:

```
void   Pl_Query_Begin        (Bool recoverable)
int    Pl_Query_Call         (int functor, int arity, PlTerm *arg)
int    Pl_Query_Next_Solution(void)
void   Pl_Query_End          (int op)
PlTerm Pl_Get_Exception      (void)
void   Pl_Exec_Continuation  (int functor, int arity, PlTerm *arg)
```

The invocation of a Prolog predicate should be done as follows:

- open a query using `Pl_Query_Begin()`
- compute the first solution using `Pl_Query_Call()`
- eventually compute next solutions using `Pl_Query_Next_Solution()`
- close the query using `Pl_Query_End()`

The function `Pl_Query_Begin(recoverable)` is used to initialize a query. The argument `recoverable` shall be set to `TRUE` if the user wants to recover, at the end of the query, the memory space consumed by the query (in that case an additional choice-point is created). All terms created in the heap, e.g. using `Mk_...` family functions (section 9.2.5, page 194), after the invocation of `Pl_Query_Begin()` can be recovered when calling `Pl_Query_End(TRUE)` (see below).

The function `Pl_Query_Call(functor, arity, arg)` calls a predicate passing arguments. It is then used to compute the first solution. The arguments `functor`, `arity` and `arg` are similar to those of the functions handling complex terms (section 9.2.1, page 190). This function returns:

- `PL_FAILURE` (a constant equal to `FALSE`, i.e. 0) if the query fails.
- `PL_SUCCESS` (a constant equal to `TRUE`, i.e. 1) in case of success. In that case the argument array `arg` can be used to obtain the unification performed by the query.
- `PL_EXCEPTION` (a constant equal to 2). In that case function `Pl_Get_Exception()` can be used to obtained the exceptional term raised by `throw/1` (section 6.2.4, page 47).

The function `Pl_Query_Next_Solution()` is used to compute a new solution. It must be only used if the result of the previous solution was `PL_SUCCESS`. This functions returns the same kind of values as `Pl_Query_Call()` (see above).

The function `Pl_Query_End(op)` is used to finish a query. This function mainly manages the remaining alternatives of the query. However, even if the query has no alternatives this function must be used to correctly finish the query. The value of `op` is:

- `PL_RECOVER`: to recover the memory space consumed by the query. After that the state of Prolog stacks is exactly the same as before opening the query. To use this option the query must have been initialized specifying `TRUE` for `recoverable` (see above).
- `PL_CUT`: to cut remaining alternatives. The effect of this option is similar to a cut after the query.
- `PL_KEEP_FOR_PROLOG`: to keep the alternatives for Prolog. This is useful when the query was invoked in a foreign C function. In that case, when the predicate corresponding to the C foreign function is invoked a query is executed and the remaining alternatives are then available as alternatives of that predicate.

Note that several queries can be nested since a stack of queries is maintained. For instance, it is possible to call a query and before terminating it to call another query. In that case the first execution of `Pl_Query_End()` will finish the second query (i.e. the inner) and the next execution of `Pl_Query_End()` will finish the first query.

Finally, the function `Pl_Exec_Continuation(functor, arity, arg)` replaces the current calculus by the execution of the specified predicate. The arguments `functor`, `arity` and `arg` are similar to those of the functions handling complex terms (section 9.2.1, page 190).

9.4.2 Example: `my_call/1` - a `call/1` clone

We here define a predicate `my_call(Goal)` which acts like `call(Goal)` except that we do not handle exceptions (if an exception occurs the goal simply fails):

In the prolog file `examp.pl`:

```
:- foreign(my_call(term)).
```

In the C file `examp_c.c`:

```
#include <string.h>
#include "gprolog.h"

Bool
my_call(PlTerm goal)

{
  PlTerm *arg;
  int functor, arity;
  int result;

  arg = Rd_Callable_Check(goal, &functor, &arity);
  Pl_Query_Begin(FALSE);
  result = Pl_Query_Call(functor, arity, arg);
  Pl_Query_End(PL_KEEP_FOR_PROLOG);
  return (result == PL_SUCCESS);
}
```

The compilation produces an executable called `examp`:

```
% gplc examp.pl examp_c.c
```

Examples of use:

```
| ?- my_call(write(hello)).
hello

| ?- my_call(for(X,1,3)).

X = 1 ?                    (here the user presses ; to compute another solution)

X = 2 ?                    (here the user presses ; to compute another solution)

X = 3                      (here the user is not prompted since there is no more alternative)

| ?- my_call(1).
{exception: error(type_error(callable,1),my_call/1)}

| ?- my_call(call(1)).

no
```

When `my_call(1)` is called an error is raised due to the use of `Rd_Callable_Check()`. However the error raised by `my_call(call(1))` is ignored and `FALSE` (i.e. a failure) is returned by the foreign function.

To really simulate the behavior of `call/1` when an exception is recovered it should be re-raised to be captured by an earlier handler. The idea is then to execute a `throw/1` as the continuation. This is what it is done by the following code:

```
#include <string.h>
#include "gprolog.h"

Bool
my_call(PlTerm goal)
{
  PlTerm *args;
  int functor, arity;
  int result;

  args = Rd_Callable_Check(goal, &functor, &arity);
  Pl_Query_Begin(FALSE);
  result = Pl_Query_Call(functor, arity, args);
  Pl_Query_End(PL_KEEP_FOR_PROLOG);
  if (result == PL_EXCEPTION)
    {
      PlTerm except = Pl_Get_Exception();
      Pl_Exec_Continuation(Find_Atom("throw"), 1, &except);
    }

  return result;
}
```

The following code propagates the error raised by `call/1`.

```
| ?- my_call(call(1)).
{exception: error(type_error(callable,1),my_call/1)}
```

Finally note that a simpler way to define `my_call/1` is to use `Pl_Exec_Continuation()` as follows:

```
#include <string.h>
#include "gprolog.h"

Bool
my_call(PlTerm goal)
```

```
{
  PlTerm *args;
  int functor, arity;

  args = Rd_Callable_Check(goal, &functor, &arity);
  Pl_Exec_Continuation(functor, arity, args);
  return TRUE;
}
```

9.4.3 Example: recovering the list of all operators

We here define a predicate `all_op(List)` which unifies `List` with the list of all currently defined operators as would be done by: `findall(X,current_op(_,_,X),List)`.

In the prolog file `examp.pl`:

```
:- foreign(all_op(term)).
```

In the C file `examp_c.c`:

```
#include <string.h>
#include "gprolog.h"

Bool
all_op(PlTerm list)
{
  PlTerm op[1024];
  PlTerm args[3];
  int n = 0;
  int result;

  Pl_Query_Begin(TRUE);
  args[0] = Mk_Variable();
  args[1] = Mk_Variable();
  args[2] = Mk_Variable();
  result = Pl_Query_Call(Find_Atom("current_op"), 3, args);
  while (result)
    {
      op[n++] = Mk_Atom(Rd_Atom(args[2])); /* arg #2 is the name of the op */
      result = Pl_Query_Next_Solution();
    }
  Pl_Query_End(PL_RECOVER);

  return Un_Proper_List_Check(n, op, list);
}
```

Note that we know here that there is no source for exception. In that case the result of `Pl_Query_Call` and `Pl_Query_Next_Solution` can be considered as a boolean.

The compilation produces an executable called `examp`:

```
% gplc examp.pl examp_c.c
```

Example of use:

```
| ?- all_op(L).

L = [:-,:-,\=,=:=,#>=,#<#,@>=,-->,mod,#>=#,**,*,+,+,',',...]
```

```
| ?- findall(X,current_op(_,_,X),L).

L = [:-,:-,\=,=:=,#>=,#<#,@>=,-->,mod,#>=#,**,*,+,+,',',...]
```

9.5 Defining a new C main() function

GNU Prolog allows the user to define his own `main()` function. This can be useful to perform several tasks before starting the Prolog engine. To do this simply define a classical `main(argc, argv)` function. The following functions can then be used:

```
int  Start_Prolog         (int argc, char *argv[])
void Stop_Prolog          (void)
void Reset_Prolog         (void)
Bool Try_Execute_Top_Level(void)
```

The function `Start_Prolog(argc, argv)` initializes the Prolog engine (`argc` and `argv` are the command-line variables). This function collects all linked objects (issued from the compilation of Prolog files) and initializes them. The initialization of a Prolog object file consists in adding to appropriate tables new atoms, new predicates and executing its system directives. A system directive is generated by the Prolog to WAM compiler to reflect a (user) directive executed at compile-time such as `op/3` (section 6.1.10, page 44). Indeed, when the compiler encounters such a directive it immediately executes it and also generates a system directive to execute it at the start of the executable. When all system directives have been executed the Prolog engine executes all initialization directives defined with `initialization/1` (section 6.1.13, page 45). The function returns the number of user directives (i.e. `initialization/1`) executed. This function must be called only once.

The function `Stop_Prolog()` stops the Prolog engine. This function must be called only once after all Prolog treatment have been done.

The function `Reset_Prolog()` reinitializes the Prolog engine (i.e. reset all Prolog stacks).

The function `Try_Execute_Top_Level()` executes the top-level if linked (section 3.4.3, page 22) and returns `TRUE`. If the top-level is not present the functions returns `FALSE`.

Here is the definition of the default GNU Prolog `main()` function:

```
int
Main_Wrapper(int argc, char *argv[])
{
  int nb_user_directive;
  Bool top_level;

  nb_user_directive = Start_Prolog(argc, argv);

  top_level = Try_Execute_Top_Level();

  Stop_Prolog();

  if (top_level || nb_user_directive)
    return 0;

  fprintf(stderr,
          "Warning: no initial goal executed\n"
          "   use a directive :- initialization(Goal)\n"
          "   or remove the link option --no-top-level"
```

```
            " (or --min-bips or --min-size)\n");

  return 1;
}

int
main(int argc, char *argv[])
{
  return Main_Wrapper(argc, argv);
}
```

Note that under some circumstances it is necessary to encapsulate the code of `main()` inside an intermediate function called by `main()`. Indeed, some C compilers (e.g. gcc) treats `main()` particularly, producing an uncompatible code w.r.t GNU Prolog. So it is a good idea to always use a wrapper function as shown above.

9.5.1 Example: asking for ancestors

In this example we use the following Prolog code (in a file called `new_main.pl`):

```
parent(bob,   mary).
parent(jane,  mary).
parent(mary,  peter).
parent(paul,  peter).
parent(peter, john).

anc(X, Y):-
        parent(X, Y).

anc(X, Z) :-
        parent(X, Y),
        anc(Y, Z).
```

The following file (called `new_main_c.c`) defines a `main()` function readinf the name of a person and displaying all successors of that person. This is equivalent to the Prolog query: `anc(Result, Name)`.

```
static int
Main_Wrapper(int argc, char *argv[])
{
  int func;
  WamWord arg[10];
  char str[100];
  char *sol[100];
  int i, nb_sol = 0;
  Bool res;

  Start_Prolog(argc, argv);

  func = Find_Atom("anc");
  for (;;)
    {
      printf("\nEnter a name (or 'end' to finish): ");
      scanf("%s", str);

      if (strcmp(str, "end") == 0)
        break;
```

```
      Pl_Query_Begin(TRUE);

      arg[0] = Mk_Variable();
      arg[1] = Mk_String(str);
      nb_sol = 0;
      res = Pl_Query_Call(func, 2, arg);
      while (res)
        {
          sol[nb_sol++] = Rd_String(arg[0]);
          res = Pl_Query_Next_Solution();
        }
      Pl_Query_End(PL_RECOVER);

      for (i = 0; i < nb_sol; i++)
        printf("  solution: %s\n", sol[i]);
      printf("%d solution(s)\n", nb_sol);
    }

  Stop_Prolog();
  return 0;
}

int
main(int argc, char *argv[])
{
  return Main_Wrapper(argc, argv);
}
```

The compilation produces an executable called `new_main`:

```
% gplc new_main.pl new_main_c.c
```

Examples of use:

```
Enter a name (or 'end' to finish): john
  solution: peter
  solution: bob
  solution: jane
  solution: mary
  solution: paul
5 solution(s)

Enter a name (or 'end' to finish): mary
  solution: bob
  solution: jane
2 solution(s)

Enter a name (or 'end' to finish): end
```

References

[1] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming Series, MIT Press, 1991. `http://www.isg.sfu.ca/~hak/documents/wam.html`

[2] W.F. Clocksin and C.S. Mellish. Programming in Prolog, Springer-Verlag, 1981.

[3] P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*, Tokyo, Japan, MIT Press, 1995. `ftp://ftp.inria.fr/INRIA/Projects/loco/publications/wamcc/wamcc.ps`

[4] P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, Vol. 27, No. 3, June 1996. `ftp://ftp.inria.fr/INRIA/Projects/loco/publications/clp_fd/long_clp_fd.ps`

[5] Information technology - Programming languages - Prolog - Part 1: General Core. ISO/IEC 13211-1, 1995. `http://www.logic-programming.org/prolog_std.html`

[6] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[7] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, 1989.

[8] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.

Index

!/0, **46**, 47
'.'/2, **136**
(',')/2, **46**
(-->)/2, **109**
(->)/2, **46**
(;)/2, **46**
(=)/2, **50**
(=..)/2, **53**
(=:=)/2, **60**
(==)/2, **51**, 122
(=<)/2, **60**
(=\=)/2, **60**
(@=<)/2, **51**
(@<)/2, **51**
(@>)/2, **51**
(@>=)/2, **51**
(#/\)/2 (FD), **174**
(#=)/2 (FD), **171**
(#==>)/2 (FD), **174**
(#=#)/2 (FD), **172**
(#=<)/2 (FD), **171**
(#=<#)/2 (FD), **172**
(##)/2 (FD), **174**
(#<)/2 (FD), **171**
(#<=>)/2 (FD), **174**
(#<#)/2 (FD), **172**
(#>)/2 (FD), **171**
(#>=)/2 (FD), **171**
(#>=#)/2 (FD), **172**
(#>#)/2 (FD), **172**
(#\)/1 (FD), **174**
(#\/)/2 (FD), **174**
(#\/\)/2 (FD), **174**
(#\=)/2 (FD), **171**
(#\==>)/2 (FD), **174**
(#\=#)/2 (FD), **172**
(#\<=>)/2 (FD), **174**
(#\\/)/2 (FD), **174**
(is)/2, **59**
(<)/2, **60**
(>)/2, **60**
(>=)/2, **60**
(\+)/1, **113**
(\=)/2, **50**
(\==)/2, **51**
--, **13**
--assembly, **22**
--aux-father, **27**
--aux-father2, **27**
--c-compiler, **23**
--cmd-line, **27**
--comment, 23, **23**
--compile-msg, **22**
--cstr-size, **23**
--encode, **27**
--entry-goal, **13**
--fast-math, **22**, 59
--fd-to-c, **22**
--fixed-sizes, 19, **23**
--foreign-only, **22**
--global-size, **23**
--help, **13**, **22**, **27**
--init-goal, **13**
--keep-void-inst, **22**
--local-size, 19, **23**
--min-bips, **23**
--min-fd-bips, **23**
--min-pl-bips, **23**
--min-reg-opt, **22**
--min-size, **23**
--mini-assembly, **22**
--no-call-c, **22**
--no-debugger, 23, **23**
--no-decode-hexa, **22**
--no-del-temp, **22**
--no-fd-lib, **23**
--no-inline, **22**
--no-opt-last-subterm, **22**
--no-redef-error, **22**
--no-reg-opt, **22**
--no-reorder, **22**
--no-singl-warn, **22**
--no-susp-warn, **22**
--no-top-level, **23**
--object, **22**
--output, **22**
--pl-state, **22**, 136
--printf, **27**
--query-goal, **13**
--relax, **27**
--statistics, **22**
--strip, **23**
--temp-dir, **22**
--trail-size, **23**
--verbose, **22**
--version, **13**, **22**, **27**
--wam-for-byte-code, **22**
--wam-for-native, **22**
-A, **23**
-C, **23**
-F, **22**
-H, **27**
-L, **23**
-M, **22**
-P, **27**
-S, **22**
-W, **22**
-c, **22**

-h, **22**
-o, **22**
-s, **23**
-v, **22**
-w, **22**

abolish/1, **63**
abort/0, 17, 32, **112**
absolute_file_name (property), **147**
absolute_file_name/2, 44, 70, 136, 137, **141**, 142, 144–149
add_linedit_completion/1, **162**
add_stream_alias/2, 67, **79**
add_stream_mirror/2, 68, **79**
alias (option), **71**
alias (property), **73**
append (mode), **70**
append/1, **107**
append/3, **121**
architecture/1, **151**
arg/3, **53**
argument selector, **126**
argument_counter/1, **142**
argument_list/1, 14, **143**
argument_value/2, 14, **143**
asserta/1, **61**
assertz/1, **61**
at_end_of_stream/0, **74**
at_end_of_stream/1, **74**
atom/1, **49**
atom_chars/2, **116**
atom_codes/2, **116**
atom_concat/3, **115**
atom_hash/2, **119**
atom_length/2, **114**
atom_property/2, **120**
atomic/1, **49**

back_quotes (flag), 14, 91, **134**, 136
back_quotes (token), **94**
backtracks (FD option), **180**
bagof/3, **66**
binary (option), **70**, 81
bind_variables/2, **56**
bip_name (option), **183**, 196
block (option), **71**, 82
block_device (permission), **147**
bof (whence), **76**
boolean (option), **183**, 184
bounded (flag), **133**
bounds (FD option), **179**
break/0, 17, 32, **112**
buffering (option), **71**
buffering (property), **74**
built_in (property), 43, **65**
built_in/0 (directive), **43**
built_in/1 (directive), **43**
built_in_fd (property), 43, **65**
built_in_fd/0 (directive), **43**
built_in_fd/1 (directive), **43**

call/1, **47**
call/2, **113**
call_with_args/1-11, **113**
callable/1, **49**
catch/3, 29, 37, **47**
change_directory/1, **144**
char_code/2, **116**, 189
char_conversion (flag), 91, 103, **134**, 136
char_conversion/2 (directive), **44**
char_conversion/2, 45, **102**
character_count/2, **76**
character_device (permission), **147**
choice_size (option), **183**, 186
clause/2, **63**
close/1, **72**
close/2, **72**, 153, 158
close_input_atom_stream/1, **83**
close_input_chars_stream/1, **83**
close_input_codes_stream/1, **83**
close_output_atom_stream/2, **84**
close_output_chars_stream/2, **84**
close_output_codes_stream/2, **84**
compare/3, **52**
completion, **18**, 162, 163
compound/1, **49**
consult/1, 16, 17, 20, 21, **136**
copy_term/2, **54**
cpu_time/1, **139**
create_pipe/2, **154**
current (whence), **76**
current_alias/2, **79**
current_atom/1, **120**
current_bip_name/2, 37, **135**
current_char_conversion/2, **103**
current_input/1, **69**
current_mirror/2, **80**
current_op/3, **102**
current_output/1, **69**
current_predicate/1, 62, **64**
current_prolog_flag/2, **134**
current_stream/1, **73**

date_time/1, **149**
debug (flag), **134**
debug/0 (debug), 17, **29**
debugging/0 (debug), **29**, 32
decompose_file_name/4, **141**
Definite clause grammars, *see* DCG
delete/3, **122**
delete_directory/1, **144**
delete_file/1, **146**
directory (permission), **147**
directory_files/2, **145**

discontiguous/1 (directive), **42**
display/1, **96**
display/2, **96**, 105, 106
display_to_atom/2, **105**
display_to_chars/2, **106**
display_to_codes/2, **106**
double_quotes (flag), 91, **134**, 136
dynamic (property), **65**
dynamic/1 (directive), **41**, 60

end_of_stream (property), **74**
end_of_term (option), **92**
ensure_linked/1 (directive), **43**
ensure_loaded/1 (directive), **44**
environ/2, **144**
eof (whence), **76**
eof_action (option), **71**
eof_action (property), **74**
eof_code (option), **71**, 81
error (option), **71**, 81, **92**
escape sequence, 14, 121, **134**
exclude (option), **56**
exec/4, **153**
exec/5, **153**
execute (permission), **147**
expand_term/2, **111**
extended (token), **94**
extra-constrained, *see* extra_cstr
extra_cstr (FD), **165**, 170

fail (option), **92**
fail/0, **46**
fct_name (option), **183**
fd_all_different/1 (FD), **176**
fd_at_least_one/1 (FD), **175**
fd_at_most_one/1 (FD), **175**
fd_atleast/3 (FD), **177**
fd_atmost/3 (FD), **177**
fd_cardinality/2 (FD), **175**, 178
fd_cardinality/3 (FD), **175**
fd_dom/2 (FD), **169**
fd_domain/2 (FD), **168**
fd_domain/3 (FD), **167**
fd_domain_bool/1 (FD), **167**
fd_element/3 (FD), **176**
fd_element_var/3 (FD), **177**
fd_exactly/3 (FD), **177**
fd_has_extra_cstr/1 (FD), **170**
fd_has_vector/1 (FD), **170**
fd_labeling/1 (FD), **179**
fd_labeling/2 (FD), **179**, 180
fd_labelingff/1 (FD), **179**
fd_max/2 (FD), **169**
fd_max_integer (FD), **165**, 166
fd_max_integer/1 (FD), **166**
fd_maximize/2 (FD), **180**
fd_min/2 (FD), **169**
fd_minimize/2 (FD), **180**
fd_not_prime/1 (FD), **173**
fd_only_one/1 (FD), **175**
fd_prime/1 (FD), **173**
fd_relation/2 (FD), **178**
fd_relationc/2 (FD), **178**
fd_set_vector_max/1 (FD), 165, **167**
fd_size/2 (FD), **169**
fd_use_vector/1 (FD), **170**
fd_var/1 (FD), **168**
fd_vector_max/1 (FD), 165, **166**
fifo (permission), **147**
file_exists/1, **146**
file_name (property), **73**
file_permission/2, **146**
file_property/2, **147**
find_linedit_completion/2, **163**
findall/3, **66**
first_fail (FD option), **179**
flag, *see* Prolog flag
float/1, **49**
flush_output/0, **72**
flush_output/1, 68, **72**
for/3, **114**
force (option), **72**
foreign/1 (directive), **45**, **183**
foreign/2 (directive), **45**, **183**
fork_prolog/1, **154**
format/2, **98**
format/3, **98**, 105, 106
format_to_atom/3, **105**
format_to_chars/3, **106**
format_to_codes/3, **106**
from (option), **56**
full (debug), **30**
functor/3, **52**

g_array (global var.), **127**
g_array_auto (global var.), **127**
g_array_extend (global var.), **127**
g_array_size/2, **128**
g_assign/2, **127**
g_assignb/2, **127**
g_dec/1, **129**
g_dec/2, **129**
g_dec/3, **129**
g_deco/2, **129**
g_inc/1, **129**
g_inc/2, **129**
g_inc/3, **129**
g_inco/2, **129**
g_link/2, **127**
g_read/2, **128**
g_reset_bit/2, **130**
g_set_bit/2, **130**
g_test_reset_bit/2, **130**
g_test_set_bit/2, **130**

generic_var/1 (FD), **168**
get/1, **108**
get0/1, **108**
get_byte/1, **89**
get_byte/2, 68, **89**
get_char/1, **85**
get_char/2, **85**
get_code/1, **85**
get_code/2, **85**, 86
get_key/1, **86**
get_key/2, **86**
get_key_no_echo/1, **86**
get_key_no_echo/2, **86**
get_linedit_prompt/1, **162**
get_print_stream/1, **100**
get_seed/1, **140**
gplc, **22**, 24, 26, 136

half (debug), **30**
halt/0, 13, 17, **112**
halt/1, **112**
hash (property), **120**
hexgplc, **26**
host_name/1, **150**
hostname_address/2, **161**

ignore_ops (option), **96**
include/1 (directive), **44**
infix_op (property), **120**
initialization/1 (directive), 24, 25, **45**, 203
input (property), **73**
integer/1, **49**
integer_rounding_function (flag), 59, **133**
interpreter, *see* top-level

jump (option), **183**, 184

keysort/1, **125**
keysort/2, **125**

largest (FD option), **179**
last/2, **124**
last_modification (property), **148**
last_read_start_line_column/2, **95**
leash/1 (debug), **30**, 31
length (property), **120**
length/2, **124**
line (option), **71**, 82
line_count/2, **77**, 78
line_position/2, **77**
linedit, **18**, 86, 162, 163
list/1, **49**
list_or_partial_list/1, **49**
listing/0, **138**
listing/1, 32, 99, **138**
load/1, 17, 21, 23, **137**
loose (debug), **30**
lower_upper/2, **116**

MA, **20**
make_directory/1, **144**
max (FD option), **179**
max_arity (flag), **133**
max_atom (flag), 119, **133**
max_depth (option), **97**
max_integer (flag), **133**, 165
max_list/2, **125**
max_regret (FD option), **179**
max_unget (flag), 88, 90, **133**
member/2, **121**
memberchk/2, **121**
middle (FD option), **179**
min (FD option), **179**
min_integer (flag), **133**
min_list/2, **125**
mini-assembly, 11, **20**, 26
mirror (option), **71**
mirror (property), **73**
mode (property), **73**
most_constrained (FD option), **179**
multifile/1 (directive), **42**

name/2, **118**
name_query_vars/2, **55**
name_singleton_vars/1, **55**, 99
namevars (option), 16, **56**, **96**
native_code (property), **65**
needs_quotes (property), **121**
needs_scan (property), **121**
new_atom/1, **119**
new_atom/2, **119**
new_atom/3, **119**
next (option), **56**
nl/0, **88**
nl/1, **88**
nodebug/0 (debug), **29**, 32
non_fd_var/1 (FD), **168**
non_generic_var/1 (FD), **168**
none (debug), **30**
none (option), **71**, 82, **183**, 184
nonvar/1, **49**
nospy/1 (debug), **30**, 32
nospyall/0 (debug), **30**
notrace/0 (debug), **29**
nth/3, **124**
number/1, **49**
number_atom/2, **117**
number_chars/2, **117**
number_codes/2, **117**
numbervars (option), 16, **56**, **96**
numbervars/1, **56**, 99
numbervars/3, **56**

once/1, **113**
op/3 (directive), **44**
op/3, 44, **100**

open/3, **70**
open/4, 67, **70**, 81, 82, 157
open_input_atom_stream/2, **83**
open_input_chars_stream/2, **83**
open_input_codes_stream/2, **83**
open_output_atom_stream/1, **84**
open_output_chars_stream/1, **84**
open_output_codes_stream/1, **84**
os_error (flag), **134**, 199
os_version/1, **150**
output (property), **73**

partial_list/1, **49**
peek_byte/1, **90**
peek_byte/2, **90**
peek_char/1, **87**
peek_char/2, **87**
peek_code/1, **87**
peek_code/2, **87**
permission (property), **148**
permutation/2, **122**
phrase/2, **111**
phrase/3, **111**
popen/3, 67, **153**
portray/1, **96**, 100
portray_clause/1, **99**
portray_clause/2, **99**, 138
portrayed (option), **96**
position (property), **74**
postfix_op (property), **120**
predicate_property/2, **65**
prefix/2, **123**
prefix_op (property), **120**
print/1, **96**, 98
print/2, **96**, 100, 105, 106
print_to_atom/2, **105**
print_to_chars/2, **106**
print_to_codes/2, **106**
priority (option), **97**
private (property), **65**
Prolog flag, 14, 36, 45, 59, 64, 88, 90–92, 94, 103, 119, **133**, 135, 136, 165, 198, 199
prolog_copyright (flag), **133**
prolog_date (flag), **133**
prolog_file (property), **65**
prolog_file_name/2, 136, **142**
prolog_line (property), **65**
prolog_name (flag), **133**
prolog_pid/1, **155**
prolog_version (flag), **133**
public (property), **65**
public/1 (directive), **41**, 61
punct (token), **94**
put/1, **109**
put_byte/1, **91**
put_byte/2, **91**
put_char/1, **88**
put_char/2, **88**
put_code/1, **88**
put_code/2, **88**

quoted (option), 16, **96**

random (FD option), **179**
random/1, **140**
random/3, **140**
randomize/0, **139**
read (mode), **70**
read (permission), **147**
read/1, **92**, 95
read/2, **92**, 95, 104, 105
read_atom/1, **93**, 95
read_atom/2, **93**, 95, 103
read_from_atom/2, **104**
read_from_chars/2, **104**
read_from_codes/2, **105**
read_integer/1, **93**, 95
read_integer/2, **93**, 95, 103
read_number/1, **93**, 95
read_number/2, **93**, 95, 103
read_pl_state_file/1, **136**
read_term/2, **92**, 95
read_term/3, **92**, 95, 103–105
read_term_from_atom/3, 14, 92, **104**
read_term_from_chars/3, **104**
read_term_from_codes/3, **105**
read_token/1, **94**, 95
read_token/2, **94**, 95, 103–105
read_token_from_atom/2, **104**
read_token_from_chars/2, **104**
read_token_from_codes/2, **105**
real_file_name (property), **147**
real_time/1, **139**
regular (permission), **147**
remove_stream_mirror/2, 68, 80, **80**
rename_file/2, **145**
reorder (FD option), **179**
repeat/0, **113**
reposition (option), **71**
reposition (property), **73**
reset (option), **71**, 81
retract/1, **62**
retractall/1, **62**
return (option), **183**, 184
reverse/2, **122**

search (permission), **147**
see/1, **107**
seeing/1, **108**
seek/4, **75**
seen/0, **108**
select/3, **122**
select/5, 68, **156**, 159, 161
send_signal/2, **156**

set_bip_name/2, 37, **135**, 196
set_input/1, 68, **69**
set_linedit_prompt/1, **162**
set_output/1, 68, **70**
set_prolog_flag/2 (directive), **45**
set_prolog_flag/2, 45, **133**
set_seed/1, **139**
set_stream_buffering/2, 68, **82**, 157, 158
set_stream_eof_action/2, **81**
set_stream_line_column/3, **78**
set_stream_position/2, 68, **75**
set_stream_type/2, **81**, 158
setarg/3, **54**
setarg/4, **54**
setof/3, **66**
shell/0, **151**
shell/1, **151**
shell/2, **151**
singleton_warning (flag), **134**, 136
singletons (option), 55, 56, **92**
size (property), **148**
skip/1, **108**
sleep/1, **156**
smallest (FD option), **179**
socket (permission), **147**
socket/2, **158**
socket_accept/3, **160**
socket_accept/4, **160**
socket_bind/2, **159**
socket_close/1, **158**
socket_connect/4, 67, 158, **159**
socket_listen/2, **160**
sort/1, **125**
sort/2, **125**
sort0/1, **125**
sort0/2, **125**
space_args (option), **96**
spawn/2, **152**
spawn/3, **152**
spy/1 (debug), **30**, 32
spypoint_condition/3 (debug), **30**, 32
sr_change_options/2, **164**
sr_close/1, **164**
sr_current_descriptor/1, **164**
sr_error_from_exception/2, **164**
sr_get_error_counters/3, **164**
sr_get_file_name/2, **164**
sr_get_include_list/2, **164**
sr_get_include_stream_list/2, **164**
sr_get_module/3, **164**
sr_get_position/3, **164**
sr_get_size_counters/3, **164**
sr_get_stream/2, **164**
sr_open/3, **164**
sr_read_term/4, **164**
sr_set_error_counters/3, **164**
sr_write_error/2, **164**
sr_write_error/4, **164**
sr_write_error/6, **164**
sr_write_message/4, **164**
sr_write_message/6, **164**
sr_write_message/8, **164**
standard (FD option), **179**
static (property), **65**
statistics/0, **138**
statistics/2, **138**
stop/0, **112**
stream_line_column/3, **77**
stream_position/2, **74**, 75
stream_property/2, **73**, 74, 75, 79, 81
strict_iso (flag), 36, 64, **134**
string (token), **94**
sub_atom/5, **115**
sublist/2, **123**
suffix/2, **123**
sum_list/2, **125**
syntax_error (flag), 92, **134**, 198
syntax_error (option), **92**
syntax_error_info/4, **95**, 198
system/1, **152**
system/2, **152**
system_time/1, **139**

tab/1, **109**
tell/1, **107**
telling/1, **108**
temporary_file/3, **149**
temporary_name/2, **148**
term_ref/2, **57**
text (option), **70**, 81
throw/1, 29, 37, **47**, 199
tight (debug), **30**
told/0, **108**
top-level, **13**, 18, 23, 25, 112, 162, 203
top_level/0, 13, **112**
trace/0 (debug), 17, **29**
true/0, **46**
type (option), **70**
type (property), **73**, **147**

unget_byte/1, **90**
unget_byte/2, **90**
unget_char/1, **88**
unget_char/2, **88**
unget_code/1, **88**
unget_code/2, **88**
unify_with_occurs_check/2, **50**
unknown (flag), **134**
unknown (permission), **147**
unlink/1, **146**
user (property), **65**
user, **107**, 108, 136, 141, 142
user_input, **68**, 72, 107, 108
user_output, **68**, 72, 107, 108

user_time/1, **139**

value_method (FD option), **179**
var (token), **94**
var/1, **49**
variable_method (FD option), **179**
variable_names (option), 55, 56, **92**
variables (option), **92**
vector_max (FD), **165**, 167, 173

wait/2, **155**
WAM, **11**, 20, 21, 32
wam_debug/0 (debug), **29**, 32
warning (option), **92**
Warren Abstract Machine, *see* WAM
working_directory/1, **144**
write (mode), **70**
write (permission), **147**
write/1, **96**, 98
write/2, **96**, 105, 106
write_canonical/1, **96**, 98
write_canonical/2, **96**, 105, 106
write_canonical_to_atom/2, **105**
write_canonical_to_chars/2, **106**
write_canonical_to_codes/2, **106**
write_pl_state_file/1, 24, **136**
write_term/2, **96**
write_term/3, 16, 31, **96**, 105, 106
write_term_to_atom/3, **105**
write_term_to_chars/3, **106**
write_term_to_codes/3, **106**
write_to_atom/2, **105**
write_to_chars/2, **106**
write_to_codes/2, **106**
writeq/1, **96**, 98
writeq/2, **96**, 105, 106, 191
writeq_to_atom/2, **105**
writeq_to_chars/2, **106**
writeq_to_codes/2, **106**