HELP POPRULEBASE A.Sloman Aug 2000 See HELP PRB_NEWS
 Revised and reorganised: Feb 2002 Revised Nov 2011

POPRULEBASE

-- Introduction ---

Poprulebase is a very flexible package supporting the construction of
single-threaded or multi-threaded mechanisms driven by condition-action
rules, which may invoke or interact with other sorts of mechanisms.
It is the core of the Sim_agent toolkit.

Poprulebase extends the programming language Pop-11 to provide a powerful
and flexible forward chaining system for specifying and running sets of
condition-action rules, including conditions and actions with interfaces
to conventional procedural programs, and also to "sub-symbolic" mechanisms
such as neural nets.

Poprulebase forms the "core" of the Sim_agent toolkit but can be used
independently of the remainder of the toolkit. The toolkit provides
further mechanisms for defining classes of interacting objects and agents,
some of which have complex internal mechanisms implemented using
poprulebase. Within the toolkit there is support for (simulated) parallel
execution of different rulesets (multi-threading), both within a single
agent, and also in different coexisting agents.

Poprulebase can be combined with the RCLIB package for creating 2-D
graphical interfaces supporting menus, sliders, dials, text input, moving
objects and various kinds of graphical interaction with running programs.
This is done in some of the graphical extensions to Sim_agent.

However Poprulebase can also be used on its own with a purely textual
interface, or in connection with other kinds of graphical interfaces
provided that the host language Pop-11 supports them or can connect to
them.

The windows version of poplog does not yet support graphics. But
poprulebase will work in it.
 An introduction to the Sim_agent toolkit can be found in a file included
with the toolkit, TEACH * SIM_AGENT, also accessible at this Web page:

 http://www.cs.bham.ac.uk/research/poplog/

 http://www.cs.bham.ac.uk/~axs/cog_affect/sim_agent.html

===

CONTENTS

```
 -- Built-in complex action types
 -- . [ADDALL [<data item>] [<data item>] ...]
 -- . [ADDIF [<pattern>] [<data item>] [<data item>] ...]
 -- . [ADDUNLESS [<pattern>] [<data item>] [<data item>] ...]
 -- . [SAY <message>]
 -- . [SAYIF <keyword> <message>]
 -- . [EXPLAIN <message>]  -- . INDATA actions [INDATA <database>
<action>]
 -- . [STOP <message>]
 -- . [STOPIF <item> <message>]
 -- . [QUIT <message>]
 -- . [QUITIF <item> <message>]
 -- . [NOT <pattern>]
 -- . [LVARS var1 var2 ...]
 -- . [VARS var1 var2 ...]
 -- . [POP11 <procedure>]
 -- . [POP11 <procedure name>]
 -- . [POP11 <pop11 instructions>]
 -- . [PUSH <item> <word>]
 -- . [POP <word>]
 -- . [DEL <integer> <integer> ...]
 -- . [DEL ?var1 ?var2 ...]
 -- . . WARNING count only SIMPLE conditions for <integer>
 -- . [REPLACE <integer> [<data item>]]
 -- . [REPLACE ?var [<data item>]]
 -- . [REPLACE [<pattern>] [<data item>]]
 -- . [MODIFY <integer> <key> <value> <key> <value> ...]
 -- . [MODIFY ?var <key> <value> <key> <value> ...]
 -- . [MODIFY <pattern> <key> <value> <key> <value> ....]
 -- . [RMODIFY <integer> <key> <value> <key> <value> ...]
 -- . [RMODIFY ?var <key> <value> <key> <value> ...]
 -- . [RMODIFY <pattern> <key> <value> <key> <value> ....]
 -- . [NULL .....]  -- . [RULE TYPE <ruletype> <name> [<conditions>]
[<actions>]]
 -- . [RULE <name> [<conditions>] [<actions>]]
 -- . SAVE and RESTORE actions (for rules or data)
 -- . Action types provided by LIB PRB_EXTRA
 -- . . [PUSHRULES <ruleset>]
 -- . . [POPRULES]
 -- . . [POPRULES <ruleset>]
 -- . . [PUSHDATA <database>]
 -- . . [PUSHDATA [<patternlist>] <database>]
 -- . . [POPDATA]
 -- . . [POPDATA [<patternlist>] ]
 -- . . [POPDATA <database>]
 -- . . [POPDATA [<patternlist>] <database>]
 -- . Action types for use with FILTER conditions
 -- . . [SELECT ?var A1 A2 ... An]
 -- . . [MAP ?var MP A1 A2 ... An]
 -- . [DOALL A1 A2 ... An]
 -- . [FAIL <message>] [WITHDRAWN JULY 1995]
 -- Manipulating dependencies [DADD ...] and [DDEL ...]
 -- Actions for tracing and interacting with the user
 -- . [PAUSE]
 -- . [READ <question> <??constraint??> <data item> <??explanation??>]  --
. . READ actions in detail
 -- . . READ action constraints
```

```
 -- . [MENU <menu> <action list> <??explanation??>]  -- . . Structure of
the <menu> vector
 -- . . Structure of the menu [<options>] list
 -- . . Structure of the menu [<mappings>] list
 -- . . The menu <action list>
 -- . . . Single action list
 -- . . . Keyed action list
 -- . . . Types of menu actions
 -- . . Example of a MENU action
 -- . . Printing out menus: prb_print_menu(message, options)
 -- User-defined action keywords and prb_action_type
 -- . Example of a new action keyword
 -- NEWLIMIT action
 -- Interactive commands available during READ, MENU or Tracing pauses  --
-- .show
 -- -- .why
 -- -- .data
 -- -- .trace and .untrace
 -- -- .show_conditions
 -- -- .chatty
 -- -- .walk
 -- -- :<Pop-11 expression>
 -- -- ? (or any unrecognised command)
 -- Actions are instantiated before they are run
 -- . Variable bindings in actions
 -- . "popval" or $$ list elements
 -- "apply" or $: list elements
 -- Rule manipulating procedures
 -- Global variables
 -- -- prb_database <the current database>
 -- -- prb_rules <a list of rules>  -- -- prb_ruleset_name <the name of
the current ruleset>
 -- -- prb_family_name: false or word
 -- -- prb_current_family
 -- -- prb_current_rule_prop
 -- -- prb_max_keys <a number>
 -- -- prb_noprint_keys <list of words>
 -- -- prb_actions_run <counter variable, or false>
 -- -- prb_trace_ruleschanged <boolean>
 -- -- this_rule <a rule>
 -- -- this_rule_name <a word>
 -- -- do_trace_match <boolean>
 -- -- word_of_ident {property}
 -- Procedures relating to the database
 -- . prb_newdatabase( hashlen, items ) -> newdb
 -- . prb_print_table( database )
 -- . prb_print_table( database, keys )
 -- . prb_print_database( )
 -- . prb_add(item)
 -- . prb_add_db_to_db(db1, db2, copying);
 -- . prb_add_to_db(item, dbtable);
 -- . prb_present(pattern) -> false or item
 -- . prb_present_keys(pattern, keys) -> item;
 -- . prb_in_database(pattern) -> false or item
 -- . prb_del1(pattern, data) -> (item, data);
 -- . prb_flush(pattern)
 -- . prb_flush1(pattern)
```

```
 -- . prb_match_apply(dbtable, pattern, proc)  -- .
prb_match_apply_keys(dbtable, pattern, keys, proc)
 -- . prb_remove_all(list_of_patterns) -> found
 -- . prb_forevery(patternlist, proc)
 -- . prb_allpresent(patternlist) -> false or found_list
 -- . prb_empty(dbtable) -> boolean
 -- . prb_storedata(<filename>)
 -- Additional procedures and variables used for tracing
 -- . prb_show_conditions (boolean or list of words)
 -- . . Example of prb_show_conditions trace output
 -- . prb_show_ruleset {boolean}
 -- . prb_chatty {boolean or integer}
 -- . prb_repeating {boolean}
 -- . prb_walk {boolean}
 -- . prb_walk_fast {boolean}
 -- . prb_pausing {boolean}
 -- . prb_explain_trace {boolean}
 -- . prb_debugging {boolean}
 -- User-definable procedures for self-monitoring
 -- . prb_checking_conditions_trace(agent, ruleset, rule);
 -- . prb_checking_one_condition_trace(agent, condition, rule);
 -- . prb_all_conditions_satisfied_trace(agent, ruleset, rule,
matchedvars);  -- . prb_condition_satisfied_trace(agent, condition, item,
rule, matchedvars);  -- . prb_doing_actions_trace(agent, ruleset,
rule_instance);
 -- . prb_do_action_trace(agent, action, rule_instance;
 -- . prb_adding_trace(agent, item);
 -- . prb_deleting_trace(agent, item);
 -- . prb_deleting_pattern_trace(agent, deleted, pattern);
 -- . prb_modify_trace(agent, item, action, rule_instance);
 -- . prb_condition_failed_trace(agent, condition, rule);
 -- . [prb_pattern_matched_trace(agent, pattern, item); ]
 -- . Controlling trace procedures: prb_tracing_on, prb_self_trace
 -- . Use of sections IFSECTIONS {boolean}
 -- . prb_rulesection {property}
 -- . prb_use_sections {boolean}
 -- . prb_check_section(rule);
 -- Additional control variables
 -- . prb_allrules {boolean, or 1}
 -- . prb_recency {boolean}
 -- . prb_useweights {boolean}
 -- . prb_sortrules {false or procedure}
 -- . prb_remember {false or list}
 -- . prb_copy_modify {boolean} (DANGER)
 -- . prb_prwarning {procedure}
 -- . prb_get_input {boolean for asynchronous input}
 -- . prb_memlim {false or integer} WITHDRAWN  -- . prb_max_conditions
{integer}
 -- . prb_add_condition_vars(list);
 -- . prb_add_action_vars(rule_instance, action);
 -- Some user definable procedures
 -- . prb_eval(action)
 -- . prb_eval_list(actions)
 -- A format for defining prb_sortrules
 -- . Selecting on the basis of specificity
 -- . Sorting on the basis of the most recent condition made true
 -- Turning tracing of individual rules on or off
```

```
 -- . prb_trace(<list of rule names>)
 -- . prb_untrace(<list of rule names>)
 -- . prb_untrace("all")
 -- Extended example: factorial
 -- Extended example TEACH * PRBRIVER
 -- Autoloadable library procedures in $poplocal/local/newkit/prb/auto  --
-- prb/auto/prb_add_list_to_db.p
 -- -- prb/auto/prb_allpresent.p
 -- -- prb/auto/prb_assoc_memb.p
 -- -- prb/auto/prb_collect_values.p
 -- -- prb/auto/prb_delete_rule.p
 -- -- prb/auto/prb_do_all.p
 -- -- prb/auto/prb_foreach.p
 -- -- prb/auto/prb_implies.p
 -- -- prb/auto/prb_interact.p
 -- -- prb/auto/prb_list_data.p
 -- -- prb/auto/prb_make_rule.p  -- -- prb/auto/prb_map_action.p
 -- -- prb/auto/prb_menu_interact.p
 -- -- prb/auto/prb_pause_read.p
 -- -- prb/auto/prb_pr_rule.p
 -- -- prb/auto/prb_print_menu.p
 -- -- prb/auto/prb_push_or_pop.p
 -- -- prb/auto/prb_read_info.p
 -- -- prb/auto/prb_remove_all.p
 -- -- prb/auto/prb_replace.p
 -- -- prb/auto/prb_rule_weight.p
 -- -- prb_ruleschanged_trace(ruleset, family);
 -- -- prb/auto/prb_save_or_restore.p
 -- -- prb/auto/prb_saverules.p
 -- -- prb/auto/prb_select_action.p
 -- -- prb/auto/prb_show_rules.p
 -- -- prb/auto/prb_trace.p
 -- -- prb/auto/prb_trace_rule.p
 -- -- prb/auto/prb_truncate.p
 -- -- prb/auto/prb_untrace.p
 -- -- prb/auto/prb_walk_trace.p
 -- -- prb/auto/prb_which_values.p
 -- Utility procedures
 -- . prb_new_rule(<name>, <weight> <conditions>, <actions>, <type>)
 -- . prb_delete_rule(<name>, <type>)
 -- . prb_pr_rule(<name>, <list>)
 -- . prb_pr_rule(<name>)
 -- . prb_forget_rules();
 -- . prb_assoc(key, list) -> val;
 -- . prb_database_keys(dbtable) -> keys;  -- . prb_is_var_key(key);
 -- . prb_in_data(pattern, data) -> item;
 -- . prb_member(item, list) -> boolean
 -- . sys_print_ident(id)
 -- . prb_variables_in(list, varlist, VARlist) -> (varlist, VARlist);
 -- . prb_extend_popmatchvars(list, matchvars) -> matchvars;
 -- . prb_valof(word) -> item;
 -- Additional procedures defined in LIB * POPRULEBASE
 -- Debugging and development aids
 -- . Use of indirection through identifiers
 -- . prb_profile
 -- . LIB prb_trace_procs (and show_trace_match)
 -- . LIB prb_checkpatterns
```

-- Latest version and "news" files ----------

Some of this information is out of date because Poprulebase is now included with Poplog by default in the $usepop/pop/packages/prb directory.

The latest version of this file is normally installed at the University of Birmingham for local users. It can also be accessed via the web here: http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/poprulebase

See HELP PRB_NEWS for changes.
The latest version will be posted at: http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/prb_news

Some related changes are in HELP SIM_AGENT_NEWS, available in the Sim_agent toolkit package.
The latest version will be posted at http://www.cs.bham.ac.uk/research/poplog/newkit/sim/help/sim_agent_news

If you are using the RCLIB package for a graphical interface you may find it useful to look at the HELP RCLIB_NEWS news file. The latest version is posted here: http://www.cs.bham.ac.uk/research/poplog/rclib/help/rclib_news

-- This file ----------

NOTE:
This help file introduces only the main features of Poprulebase, giving cross references to additional features described elsewhere.

A basic tutorial introduction is in TEACH RULEBASE, included with the package and accessible from inside the editor. It is also available on the web at http://www.cs.bham.ac.uk/research/poplog/newkit/prb/teach/rulebase

A more advanced tutorial is in TEACH POPRULEBASE, available on the web at
http://www.cs.bham.ac.uk/research/poplog/newkit/prb/teach/poprulebase

Many features are also introduced in teach files which form part of the Sim_agent library, e.g. TEACH SIM_FEELINGS

NOTE:
If you are running pop-11 or ved and wish to make the help, teach, and library files available, first do

```
uses newkit
```

which puts all the toolkit directories onto relevant search lists.

As an alternative if you are interested only in poprulebase, then it is possible that this command has been made available at your site:
 uses prblib

-- More detailed explanation --
 This package makes available a forward-chaining condition-action rule interpreter. It allows the user to define multiple sets of condition-action rules, and multiple databases and to run the rules in connection with different databases. The rules can switch to other databases or other rulesets, and there are actions which "push" or "pop" databases or rulesets.

An important feature is that different control mechanisms can be used in connection with different rulesets. For example the [DLOCAL ...] expression, defined below, makes it possible for one ruleset to employ a conflict resolution strategy selecting only one runnable rule on each cycle of the interpreter while another ruleset allows every runnable rule to have all its actions executed on each cycle. This mechanism also provides a powerful tool for selective debugging.

Poprulebase is implemented in Pop-11, an interactive, incrementally compiled, AI language, forming part of the Poplog system described at the following Web sites: http://www.cs.bham.ac.uk/research/poplog/poplog.info.html ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/poplog.info.html

 http://www.cs.bham.ac.uk/research/poplog/userguide.html

and also
 http://www.cogs.susx.ac.uk/users/adrianh/poplog.html http://www.cogs.susx.ac.uk/users/adrianh/pop11.html http://www.cvg.cs.reading.ac.uk/poplog

Poplog is available free of charge from this site

 http://www.cs.bham.ac.uk/research/poplog/freepoplog.html
or
 ftp://ftp.cs.bham.ac.uk/pub/dist/poplog/freepoplog.html

There is a mirror site at http://www.poplog.org

Development and testing using this sort of system can be very much faster than it would be if the rules were coded in languages like C, C++ or Java.

-- -- Levels in poprulebase: rules, rulesets, rulefamilies, rulesystems

In order to support the mechanisms of the SIM_AGENT library, Poprulebase supports the notion of four levels of complexity.

1. First of all is the notion of a rule. This contains a set of conditions, and a set of actions. The conditions may be simple
 or complex, as described below. When all the conditions are satisfied the rule is runnable. Satisfaction of conditions is mainly checked against the contents of the current database, though there are also more

complex types of conditions.

2. Secondly there is the notion of a ruleset. A ruleset is a list of rules. In the simplest uses of poprulebase, the main rule interpreter, prb_run is given a list of rules and a database, and it repeatedly runs the rules until either some specified limit has been reached, or a "STOP" action (described below) has occurred.
By default the order of the rules in a ruleset determines the order in which their conditions should be tested. By default this order also determines which rules run if several have all their conditions satisfied.

3. At the next level is a rulefamily, sometimes referred to as a rulecluster. A rulefamily is a special Pop-11 datastructure combining a collection of different rulesets. Only one ruleset in the family is ever "in control", i.e. the current one, at a time, but the current ruleset can transfer control to another ruleset in the same family.

This allows a complex rule based system to be divided into a set of different rulesets, such that only one relatively small ruleset is active at any one time. This provides advantages of speed and modularity, consistent with the sophisticated variety of condition types available, making automatic indexing of rules difficult.

The rulefamily concept is particularly important in connection with the SIM_AGENT library. See HELP * SIM_AGENT and the demonstration in TEACH * SIM_DEMO (after doing "uses simlib" if necessary.)

4. A still higher level is available for use in connection with the SIM_AGENT library. This is the rulesystem, which is a collection of rulefamilies and rulesets, to be run in simulated parallelism, representing a collection of coexisting concurrent mechanisms inside an agent or machine.
For more on this see HELP RULESYSTEMS

-- -- Flexibility

The Poprulebase library is written in such a way as to be tailorable by knowledgeable users, who may wish to alter parts of the code.

A collection of global control variables and user-definable procedures makes a wide variety of default behaviours possible, including different strategies for "conflict resolution", i.e. deciding which rule or rules should run when there are several whose conditions are satisfied.

This is extended in the design of the sim_agent library, where Object Oriented Programming methods based on Pop-11 Objectclass are used to support tailorability.

-- -- Why use Pop-11?

Pop-11 (like Lisp, Prolog and other AI languages) provides automatic garbage collection, run time type checking, unlimited precision arithmetic, object-oriented programming, user-definable error handling, and other features that facilitate the design of very complex programs,

especially those for which exhaustive analysis and testing in advance is theoretically or practically impossible. These features also support powerful facilities for end-user extensions to systems.

Many expert system shells provide more advanced user interfaces and are easier to use than Poprulebase, provided that the class of problems for which they were designed includes the user's problem.

Poprulebase aims to be more general and flexible. The variety of forms for conditions and actions is unusual, including not only full access to the underlying general purpose language (Pop-11), and other 'external' languages, in both conditions and actions but also condition forms such as these three:

 [NOT_EXISTS [male ?x] [adult ?x] [NOT teacher ?x]]
 True if no value for x exists such that all three conditions
 are satisfied in the current database,
  [IMPLIES [[male ?x] [adult ?x]] [teacher ?x]]
 An alternative form of the above, possibly easier to understand.

 [ALL ?<variable>]
 Here the value of the variable is a list of conditions found
 in the database - i.e. meta-conditions.

 So if the variable "constraints" has been bound to a list of lists, each of which is a condition that poprulebase understands, then
 [ALL ?constraints]
 is a condition that checks that all the constraints are satisfied. If variables in the conditions are bound by the
 check they may be used in subsequent conditions or actions.

 An example occurs in TEACH PRBRIVER, where constraints are defined in the database, and interpreted by a meta-rule.

User-defined condition types and action types can be added.

There are also several different control strategies available, and it is possible to switch between control strategies, using the
 [DLOCAL ....] construct, for instance when poprulebase is used as part of the SIM_AGENT toolkit, in which agents have internal mechanisms consisting of multiple rulesets. (See HELP SIM_AGENT)

The [DLOCAL ...] construct in Poprulebase allows contexts to be switched for individual rules or for whole rulesets. Thus tracing for a particular rule can easily be turned on or off. This is modelled on and makes use of the powerful "dlocal" mechanism for "dynamic local expressions" in Pop-11, described in HELP DLOCAL

-- -- Introductory material

For someone who has not previously programmed a rule based system it may be very hard to digest all the information in this file. For such people it may be a good idea to read these files first:

 TEACH RULEBASE
 A very basic introduction

TEACH POPRULEBASE
Gives more detail with more complex examples.

For an explanation of the use of poprulebase within the Sim_agent toolkit see
TEACH SIM_AGENT
See the section on related online documentation at the end of this file.

-- Recent changes --

MAJOR CHANGE IN JULY 2000
The new format for conditions and actions [INDATA ?db [....]]
allows conditions or actions to be applied to an alternative
database.

THERE WERE MAJOR CHANGES IN JULY 1999
Version 5.0 of poprulebase was introduced, including several changes designed to support changes in sim_agent allowing agents to get better access to the contents of their rulesystems and rulesets, etc. The details are in the HELP NEWKIT document which is part of Sim_agent.

Previously there were major changes in June 1996
The "rulefamily" and "rulesystem" mechanisms were introduced, along with other changes described by HELP * PRB_NEWS and HELP * RULESYSTEMS

-- Making the library available ----------------------------------

If you have the Sim_agent toolkit, then the command

uses newkit

will make LIB poprulebase available, along with other libraries.
Other possibilities may work, depending on how the poprulebase library has been installed. It can be installed without the rest of the toolkit in a directory tree starting prb/ e.g.

$poplocal/local/prb/

Or if it is part of newkit it may be

$poplocal/local/newkit/prb/

Often the first of those is a symbolic link to the second.

In either case the prb directory should contain a startup file
prblib.p

If that file has been linked to any of the directories in popuseslist, e.g. to $poplocal/local/lib

Then this command will work:

uses prblib
To extend search lists such as vedhelplist, popautolist, popuseslist, etc. to include the directories used by the poprulebase libraries. After this command it is possible

to do HELP POPRULEBASE, HELP PRB_NEWS, etc.

Alternatively if the poprulebase system has been installed in $poplocal/local/prb/

Then do

load $poplocal/local/prb/prblib.p

Once that has been done, search lists are set up so that a variety of USES, LIB, TEACH, and HELP commands will work.

The first command is essential if anything is to be done:

uses poprulebase
To compile the core file and make the facilities ready for use, e.g. the facilities for defining rulesets and also the main procedure, prb_run.
(This is done automatically by uses sim_agent)

uses prb_profile
To provide a profiler for recording which rules are used most, extending the Pop-11 profiler described in HELP PROFILE

uses prb_trace_procs
To provide extended tracing of rules and conditions, as described in HELP PRB_TRACE_PROCS

uses prb_filter
This makes available FILTER conditions (e.g. for running a neural net from a condition or an action), as described in
HELP * PRB_FILTER.

uses rulefamily
This makes available a "rulefamily" mechanism, described in
HELP * RULESYSTEMS

At some sites (e.g. Birmingham) there may be a saved image prb.psv that includes poprulebase. In that case the saved image may be invoked with the following Unix shell command

pop11 +prb

There may be other local saved images that include poprulebase, e.g at Birmingham the sim_agent toolkit includes Poprulebase, so the command

pop11 +sim

will also make poprulebase available. See TEACH * SIM_AGENT

If you need to set up such saved images to avoid having to recompile the toolkit each time you start, consult your local poplog administrator or post a query to the comp.lang.pop newsgroup.

-- Historical note: Poprulebase and its predecessors ------------------

The Poplog Pop-11 system has at various times included various more or less sophisticated rule-based system shells, primarily provided for teaching purposes, but with the possibility of serious applications.

Early versions are described in
 TEACH PSYS
 TEACH PRODSYS
 TEACH EXPERTS
 Note this refers to LIB ESHELL, which has a serious bug fixed
 at Birmingham. The fixed version is available as ftp://
ftp.cs.bham.ac.uk/pub/dist/poplog/lib/eshell.p

Around 1990 I started work on a more sophisticated rule interpreter which eventually found its way into the Poplog library as LIB NEWPSYS and is described in Poplog systems in the files HELP NEWPSYS and TEACH PSYSRIVER. Poprulebase supersedes this in many ways, including efficiency when databases are large.

Eventually it became clear that Newpsys had a number of flaws and restrictions, and while at Birmingham I started work on a new version, known as POPRULEBASE, intended for use with the SIM_AGENT library. The first version because available in October 1994.

Compared with LIB NEWPSYS, Poprulebase has various extensions making it easier to control and allowing more facilities for tracing, and for combining rule-based techniques with other techniques, e.g. neural net mechanisms, in a hybrid system.

For example, the "FILTER" conditions, described in HELP * PRB_FILTER enable individual rules to invoke a neural net or other "subsymbolic" mechanism to decide how to combine conditions in a rule. The DLOCAL forms described below allow flexible switching between alternative control strategies for the rule interpreter.

Poprulebase is also the core of the SIM_AGENT toolkit available from the Birmingham Poplog ftp directory. This is Pop-11 package for simulating multiple interacting agents with complex internal processing.

There are now so many facilities in Poprulebase that it is difficult to take them all in at once. New users may therefore prefer to start with TEACH * RULEBASE, which gives simplified explanations and examples. See the other tutorial files listed at the end of this file.

There were major enhancements in July 1995, after it had been used for several months in Birmingham and at DRA Malvern. The main change was the replacement of the simple list format for the database. Instead a property is now used, so that database items are indexed by their first element. (More sophisticated indexing may be added later. See the sections on efficiency and representation, below.)

Between January and June 1996 several additional major enhancements were introduced of which the most important was changing the compilation procedures so that by default all pattern variables are now lexically scoped, as described in HELP RULESYSTEMS. (This was made possible by a change in the Pop-11 procedure valof in version 15.0 of Poplog, which meant that the pattern matcher would work on patterns containing identifiers rather than words.) A number of intermediate releases were

made available at Birmingham.

Further information about changes can be found in
 HELP PRB_NEWS
 HELP SIM_AGENT_NEWS

-- Poprulebase and the Pop-11 database ------------------------------

Although this package uses the Pop-11 pattern matcher, the database used by this package is different from the main Pop-11 database package using these procedures and syntax forks:

 database, add, remove, lookup, allpresent, foreach, forevery etc.

(See HELP * DATABASE)
 Poprulebase uses a different sort of database, based on a Pop-11 property, in which all database items are indexed by their first element.

Thus the normal Pop-11 database can be used in addition, if required and will not interact with the poprulebase databases. The identifiers in the Poprulebase package all have names starting with 'prb_'.

Another major difference is that Poprulebase supports matching of conditions whose variables are lexically scoped whereas the main Pop-11 database library uses variables that are inherently global, as does LIB NEWPSYS. The change to lexical scoping considerably reduces the risk of undesirable interactions between programs.

A more detailed explanation of the similarities and differences between the pop-11 database and the Poprulebase database, and how to translate between them can be found in
 HELP PRB_DATABASE

See HELP LEXICAL, and HELP RULESYSTEMS, which gives further information on the scoping of pattern variables.
 HELP DOESMATCH provides information on the extensions to the pop-11 pattern matcher to allow it to use lexically scoped variables, along with the "!" pattern prefix.

-- Note on the use of the matcher -----------------------------------
(Skip this section on first reading)

Extensive use is made of the Pop-11 pattern matcher. See HELP * MATCHES.

There are two versions of this, both of which use the global variable popmatchvars to control which variables have been found in patterns and must not be given new values during a match. The procedure -matches- defines popmatchvars as a local variable and therefore after it returns one cannot tell which variables were set, whereas the procedure -sysmatch- sets popmatchvars and does not restore it.

Most of the poprulebase procedures use sysmatch, and may therefore alter popmatchvars. If you try to do sophisticated things with the pattern matcher this can cause you problems. However, using the procedures defined here and the system operator "matches" should always work as expected.

The use of the Pop-11 matcher originally meant that the variables used in conditions and actions had to be essentially global Pop-11 variables.

In version 4.0 this was changed by the introduction of a new default for pattern variables. They are now lexically scoped, as explained in
 HELP * RULESYSTEMS

-- RUNNING POPRULEBASE
-- -- The top level interpreter prb_run, or prb_run_with_matchvars

LIB POPRULEBASE provides syntax forms for conveniently defining condition action rules, and includes a forward-chaining production system interpreter, called prb_run, which is invoked with two arguments,

 1. a rulefamily or a ruleset (list of rules),
 or the name of one (a word)

 and

 2. an initial (possibly empty) working memory: a property or a list of lists.

It also allows a third optional "limit" argument

 3. a number, or false

to limit the number of cycles of the interpreter. Otherwise it runs until one of the rules executes a "stopping" action, explained below. If the limit is 0, that is equivalent to giving false as value. I.e. there is no cycle limit.

A slightly different procedure is called prb_run_with_matchvars, for use in connection with the SIM_AGENT library. The main difference is that the latter does not set popmatchvars to [] when it starts. That means that an environment for matcher variables can be set up in advance.

Also the third argument is NOT optional when prb_run_with_matchvars is used.

-- -- Formats for prb_run

It can be run in one of the following formats, where the rules argument is either a ruleset or a rulefamily, or a word whose valof is a ruleset or rulefamily.

 prb_run(rules, list, limit);
 prb_run(rules, property, limit);
or
 prb_run(rules, list);
 prb_run(rules, property);

-- -- Formats for prb_run_with_matchvars

If popmatchvars is not to be set locally to [] use:

 prb_run_with_matchvars(rules, list, limit);

prb_run_with_matchvars(rules, property, limit);

-- -- What the rule interpreter does

prb_run is a very simple procedure which sets up popmatchvars, ensures that there are three arguments available for prb_run_with_matchvars which it then runs.

When prb_run_with_matchvars is invoked it checks whether the second argument is a list, and if so converts it to a property. That property or the one provided is then made the value of the global variable prb_database, which is dynamically local to the procedure. I.e. its previous value is re-set after the procedure terminates. (This allows an action in a rule to invoke poprulebase with a different database.)

If the rules argument is a word, the word is saved, and its (recursive) valof is assigned to rules, then later to prb_rules. That check and assignment is now (since November 24 2011) repeatedly done within the main loop of the rule interpreter, allowing a ruleset created by
 define :ruleset <name>

to be recompiled, as long as <name> is not prb_rules.
 After setting up the database, the interpreter checks whether the rules argument is a ruleset or a rulefamily. If it is a rulefamily it selects the "current" ruleset in the family and interprets that. Any inefficiency introduced by this mechanism (which significantly aids debugging) can be removed by giving prb_run the actual ruleset rather than its name.

-- -- . Switching rulesets

It is possible for a rule action to switch to a different ruleset, which is used thereafter until the current ruleset is changed again, or the interpreter terminates.

If switching happens with a rulefamily, the rulefamily data-structure will remember which ruleset in that rulefamily is current, so that the rulefamily can be re-run later, starting off with the latest ruleset selected as current. (This is essential for the use of Poprulebase within Sim_agent.)

Switching is described in more detail below.

-- -- . Interpreting a ruleset, using different control strategies
 Having determined the ruleset to use, the interpreter repeatedly cycles through the rules in that ruleset trying to find one or more rules whose conditions are all satisfied in a consistent way. For a given rule it may find several different ways of matching its conditions against the database.

Having found a way of making the conditions true it constructs a rule-instance which specifies values for condition variables. The actions of the rule-instance can then be run with those values.

Whether only the first rule-instance found is run, whether they are all run, whether they are run in the order found or in some different order, are all depending on control strategies described below. These control strategies can be set to be different for different rulesets within a

rulefamily or a rulesystem, providing enormous flexibility. The control strategies use the global variable prb_allrules and the user-definable procedure prb_sortrules.

So on each cycle of the interpreter it goes through the rules in the current ruleset, in the order in which they occur in the ruleset creating and running rule-instances in accordance with the specified control strategy. (The default is to find the first rule-instance and run that, then start the next cycle, because by default prb_allrules is false).

Possible forms of conditions and actions are described below.

-- -- . When the interpreter stops

The interpreter will stop if it finds no rules with conditions satisfied.

It does this either until a
"stopping" action is invoked, or the number of interpreter cycles has exceeded the limit, if the limit is an integer.

It then chooses one or more of the "rule-instances"
corresponding to the satisfied conditions and runs them, in accordance with strategies provided by the user. Examples are provided in TEACH POPRULEBASE and TEACH PRBRIVER

If the rules argument is a rulefamily structure, this is composed of several rulesets, only one of which is current at any time, though control can be transferred between the component rulesets. Between calls of prb_run with the same rulefamily, it will remember which ruleset is the "current" one.

While prb_run is active it uses the global variable prb_database to hold the database against which the rule conditions are checked. If the second argument of prb_run is a property (previously created using prb_newdatabase) it is assigned locally to prb_database.

If the second argument to prb_run is a list it is used to initialise a property (see HELP * NEWPROPERTY) which stores the items in separate sublists for efficiency. For example, if the list given is

```
 [[on block1 block2] [on block2 block3] [colour block1 green]
 [size block1 big] [colour block2 red] [size block3 small]]
```

then these items will be sorted into three lists, where each list has items starting with a common first item. Then the property prb_database will associate the word "colour" with the list of lists

```
 [[colour block1 green] [colour block2 red]]
 the word "on" with the list of lists

 [[on block1 block2] [on block2 block3]]
```

and the word "size" with the list of lists

```
 [[size block1 big] [size block3 small]]
```

This means that by carefully choosing the format of information stored in the database the user can avoid slow searches down very long lists. (These examples use the words "colour" "on" and "size" as keys. The keys do not need to be words, however. They can be any constant items, e.g. numbers, lists, etc. However if the key is a datastructure such as a list or string or vector, remember that unless two keys are equal according to "==" they will be treated as different. See HELP * EQUAL) so these two items will not have the same key

```
 [[directly above] A B]
 [[directly above] B C]
```

whereas using a fixed list for the first element would overcome this as in this example:

```
 vars above_key = [directly above] ;

 [^above_key A B]
 [^above_key B C]
```

The last two items would be stored in the same list in the property prb_database.
 -- -- Recording the rules and database at the end of the run

The procedure psys_finish, described below, can be used to interrogate the state of the database and the ruleset when the procedure is about to exit.

-- What is a rule? ----------

Each rule is defined by
 1. a list of conditions to be tested against the contents of the working memory (i.e. the PRB database), and

 2. a list of actions to be performed if all the conditions are satisfied.

However, even if all the conditions are satisfied the actions may or may not be performed. That is because some other rule whose conditions are satisfied might be given priority.

In addition a rule may specify a weight, which can be used in determining priority when more than one rule has all its conditions satisfied. The weights may be used either in the obvious way as a priority indicator for sorting applicable rules or by user programs in some non-obvious way (e.g. treating weights in different ranges differently.)

For more details see the sections on prb_useweights and prb_sortrules.

Some rules also have a rule-type. Rules of the same type are stored in a list of rules, known as a "ruleset". The ordering of the rules in the list depends on the order in which the rules were initially compiled.

The standard format for defining a ruleset is now explained.

-- Defining rulesets (define :ruleset) ----------

The basic format for defining a ruleset is

```
 define :ruleset <name>;
 [DLOCAL <vars spec>]; ;;; optional
 [LVARS <popmatchvar spec>]; ;;; optional
 [VARS <popmatchvar spec>]; ;;; optional
 use_section = <boolean>; ;;; optional
 debug = <boolean>; ;;; optional

 RULE <name>
 <conditions>
 ==>
 <actions>

 vars .... ; ;;; optional

 RULE <name>
 weight <integer> ;;; optional
 <conditions>
 ==>
 <actions>

 ....  enddefine;
```

Note: instead of "RULE" for rule headers "rule" is acceptable.

Also instead of "==>" it is possible to use any word that is in the list prb_condition_terminators, which, by default includes only
 ";", "==>" and "-->".

This format for defining a ruleset allows VED's commands on current procedures to be used, e.g. <ENTER> mcp, jcp, and lcp.
 (See HELP * MARK).

For examples see TEACH RULEBASE, TEACH PRBZOO, TEACH PRBRIVER

-- . Specifying a weight

If the weight <integer> is not specified it defaults to pop_min_int. The weights can be used to determine relative priorities of rules. See prb_useweights, below.

-- . The conditions/actions separator can be changed

The symbol "==>" is used as the separator between <conditions> and <actions> in the format given above.

However, the syntax is flexible in that users can specify that the separator should be some other symbol. In fact any word in the user assignable list -prb_condition_terminators- is acceptable, and the default value contains ";", "-->" and "==>". So for example, the format

```
 RULE <name> <conditions> --> <actions>
```

will also work.

-- . Poprulebase can invoke Pop-11

Both to enable communication between rules and Pop-11 programs, and also to enable some things to be done more efficiently, the library allows both conditions and actions to include pre-compiled Pop-11 instructions, e.g. for doing extra checks on satisfied conditions, or to perform actions that have side-effects on Pop-11 structures other than the database. The main formats for this are conditions of the form:

 [WHERE <pop11 expression>]

and both conditions and actions of the form

 [POP11 <pop11 expression>]

Both are explained in more detail below.

In addition the [VARS...] and [LVARS...] forms can invoke Pop-11 procedures.

Sub-expressions of the following forms can also invoke Pop-11 procedures

 [popval <pop11 expression>]

and
 [apply <procedure> <arg1> <arg2> ...]

These can be abbreviated as
  [$$ <pop11 expression>]
and
 [$: <procedure> <arg1> <arg2> ...]

These forms, which can be embedded in conditions and actions are evaluated and their results spliced into the condition or action before the condition or action is used. Both are explained below.

NOTE 1: since the introduction of lexically scoped pattern variables in Version 4.0 the expression in the [popval ...] form is pre-compiled when rules are read in, so that variables can be shared between these expressions and the other conditions and actions in the same rule. This means that "?" and "??" should not be used with such variables.

NOTE 2: Since poprulebase rules can invoke Pop-11 procedures, and Pop-11 can invoke prolog or other Poplog languages, this implies that poprulebase can also invoke these other languages. It can also invoke external languages, such as C, Pascal, or Fortran, as described in
 HELP EXTERNAL

-- What the interpreter does --
 On each cycle of the interpreter, all rules in the current ruleset that have all their conditions satisfied are found (or, if the variable prb_allrules has the value FALSE, then only the first such rule is found).

For each such "applicable" rule a rule-instance is created (in which rule variables have been bound).

The list of possible rule instances is given to a user-definable selection procedure to determine which rules should have their actions executed on that cycle.

The procedure used to check whether all conditions of a rule are satisfied, and to create instances of those rules is the most complex procedure in the package: prb_forevery

On each cycle, after applicable rules have been found and rule-instances created, the procedure prb_do_rule is applied to selected rule-instances to "run" their actions, which may change the database or do more complex things, including interacting with the user, or running arbitrary Pop-11 procedures.
 There are several global control variables that control the search strategy, the amount of trace printing, and the interaction with the user, including a limited "explanation" facility. For example, it is possible either to specify that all rule activations should be traced interactively, by setting the global variable prb_walk TRUE, or else to selectively trace or untrace individual rules if prb_walk is FALSE.

There are also dialogue management facilities in the form of "MENU" actions for interacting with the user. By default these are purely text based, but in principle they could be combined with the Birmingham Pop-11 library for manipulating pop-up menus and graphics.
(See TEACH POPUPTOOL)

A typical invocation of the system would first define a collection of rules (using the syntactic form described below) held in a list, and set up an initial working memory and then give the command:

 prb_run(rules, data)
or

 prb_run(rules, data, N)
 where N is some integer chosen to limit the number of cycles to be allowed. (Note the lists could be any list

It is possible to have additional lists of rules, and some of the rules may perform actions that switch from the current ruleset to another ruleset. (For more on this see HELP PRB_EXTRA).

The package provides a wide range of facilities for testing and debugging, and for tailoring the behaviour to different requirements, including limiting the size of the working memory, varying the strategy for selecting the most appropriate rule if several are applicable, allowing all applicable rules to run on each cycle, giving "canned" text for explanations, preventing the same rule from being used more than once if satisfied on the same data, allowing simple backtracking, and so on.

-- . prb_finish(rules, database)

The user-definable procedure prb_finish is invoked by prb_run on completion. It is given the current values of prb_rules and prb_database as arguments. So it can be used to print out the final value of the working memory, or perhaps store it in a file, etc. An example using prb_match_apply is given below.

Usually prb_rules will be no different from the original list given to prb_run, but in principle it is possible for programs to modify it, so the final value is also made available to prb_finish.

-- Further features --

There is a rich collection of built-in action types for use in rules, including the REPLACE action, and actions PUSH and POP for convenient manipulation of stacks in the working memory. In addition users can define their own action types, or else directly invoke Pop-11 procedures in actions.

There is a simple default syntax procedure for defining rulesets. However, the mechanisms are extendable to use a different syntax or do far more checking.

There is support for hybrid systems combining rule-based and other mechanisms. In particular, the "FILTER" conditions, described in the file HELP * PRB_FILTER, enable individual rules to invoke a neural net or other "subsymbolic" mechanism to decide how to combine conditions in a rule.

In order to get an overview of the mechanism it may be useful to look at the simple examples in TEACH * RULEBASE, the more detailed examples in TEACH * POPRULEBASE and the complex planning program illustrated in TEACH * PRBRIVER, before reading the detailed description that follows.

-- Note on efficiency and representation ------------------------------

Because the conditions are checked using the standard Pop-11 pattern matcher (or something similar if that is later replaced) the process of matching a simple condition against a database item proceeds from left to right. Thus it is not desirable to start a condition with a variable as that will match everything (unless a restriction procedure is used, as described in HELP * MATCHES).
 This is one example of the general point that careful thought about the syntax used for database items and condition patterns can make a great difference to the efficiency of a program.

The internal database is a property table (a form of hash table) that uses the first element of each database assertion as a key. i.e. all assertions starting "plan" are in stored in one place, while all assertions starting with "belief" are in placed elsewhere. Thus the searching procedures, when given a pattern, use the first element of the pattern to index the hash table. (The first element does not need to be a word: it can be any sort of Pop-11 object that is uniquely identifiable.)

If a pattern starts with a variable (e.g. "?item" or "??item") or a don't care symbol (i.e. "=" or "==") then there is no fixed first item that can be used to index the property and extract a list to be searched. So the system has to search along ALL the lists, looking for items that match. Thus using such patterns as conditions in rules will lose the efficiency advantages of the partitioned database, unless such a pattern occurs as a condition in a context where the first variable will already have been given a value in an earlier condition. In that case the new condition can be instantiated before it is matched against the database, leading to a far more efficient search.

Another thing that can make a difference is to divide the program up into different sets of rules, and different databases, and run them separately. That's because if there are R rules (within the same ruleset), with on average C conditions, and D database items which are all kept in one list, finding which rules have their conditions satisfied takes a time roughly proportional to R*C*D.

To some extent this is alleviated by dividing each database into different lists depending on the initial element of each item. In that case D can be thought of as the average number of database items sharing the same first element.

Facilities are provided to support partitioning of databases and rule sets, which helps further. (E.g. the PUSHRULES, POPRULES, PUSHDATA, POPDATA and PUSH and POP actions.)

A little thought in constructing rule conditions and the use of initial keys for database items, will ensure that the automatic partitioning of the database (using the newproperty facility in pop11) will result in a much faster run times.

See also HELP * EFFICIENCY, and the notes below on using POP11 conditions with extra variables instead of WHERE conditions.

-- Adding extra checks to the rule compiler ----------------------------

There are two user-definable procedures used in reading in rule definitions. They are used to read in the list expressions defining conditions and actions, respectively:

 prb_readcondition() -> list;
 The default version is defined in LIB * POPRULEBASE. If you wish to change it copy that version and modify it as needed. It mainly uses listread to read in conditions, but it checks for special cases such as [POP11 ...] conditions and [WHERE ...] conditions and pre-compiles them.

 prb_readaction() -> list;
 The default version calls -listread- and returns the result unless it starts with "POP11" in which case, if necessary, it compiles the procedure in the tail of the list. See POP11 actions below. It is best to look at the original if you
 intend to redefine it.
 (SHOWLIB * POPRULEBASE/prb_readaction for full details)

Users can redefine these to do extra syntax checking in rule definitions, e.g. if all conditions and all actions have to have certain formats.

At present they do no checking, apart from requiring a list expression for each condition or each action. Because the internal structure of the lists is not checked, errors (e.g. wrong format for a "MODIFY" action) will be detected only at run time. Checking action formats can be achieved by re-defining -prb_readaction-.

A possible extension of this package would be to redefine these two procedures to allow an English-like notation for expressing rules or actions. (Other languages could be used instead of English.)

WARNING: anyone who tries to redefine these procedures should look carefully at how they work, including how they deal with special cases such as POP-11 procedures.

-- Data types in poprulebase: rule records and instance records -------

-- . The format of rule records:

Each rule is a record of type "prbrule", with the following fields:

 prb_rulename
 A word. Note that the rule is not the valof the word. Rather
 The rule is associated with the word in the list corresponding to the rule type. The same name can be used for different rules in different rule types.

 prb_weight
 A number, used for prioritising rules if prb_useweights is true.

 prb_conditions
 A list of conditions, of formats described below.
  prb_actions
 A list of actions, of formats described below

 prb_ruletype
 A word, whose value will be a list of rules.

 prb_rulevars
 A list of the variables used and set in the conditions and actions. (Not yet used. May be withdrawn)

(It may be necessary later to associate a Pop-11 section with a rule: see REF * SECTIONS. Draft mechanisms for this exist, turned on by making prb_use_sections true. But this may change.)

The constructor consprbrule can be used to construct such rules, and isprbrule to recognize them. However, users, should not need to access these procedures directly.

Instead the user should employ either the procedure

 init_prb_rule(name, weight, conditions, actions, type, rulevars, create)

or

 prb_new_rule(name, weight, conditions, actions, type, rulevars)

defined in LIB * POPRULEBASE

-- Defining a rule separately from a ruleset --------------------------

Normally a rule will be defined within a ruleset, using the format described above:

 define :ruleset ....

```
 ...
 enddefine;
```

The following syntactic form can be used to add a single rule to an existing ruleset:

```
 define :rule <name> in <rulesetname>
 <conditions>
 ;
 <actions>
 enddefine;
```

(See also HELP * OLDRULESYNTAX)

This reads the <name>, the <conditions> and the <actions> checking that the latter are all lists, and puts the new rule at the end of the list specified by <name> (which defaults to "prb_rules". If there is already a rule with this name in the ruleset, then its conditions and actions are replaced by the new ones. Thus this syntax makes it easy to edit and recompile a rule (e.g. unlike the library package LIB PRODSYS).

If there is already a rule with the <name> then this replaces its conditions and actions. If there isn't a new rule is appended to the ruleset.

However, normally the "define :rule..." format will not be used as the define :ruleset ... enddefine format described above is more convenient.

-- . The format of rule instances:

When a rule is found to be applicable, because all its conditions are satisfied, a rule instance, or rule activation, record is created. This is a record of type "prbactivation". Each rule instance record has the following fields:

prb_ruleof
The rule record, an object of the type described above. prb_varsof
A list of words occurring as variables in the conditions of the rule.
prb_valsof
A list of the values bound to the variables as a result of matching all the conditions to establish applicability. prb_foundof
A list of the items in prb_database (the working memory) found to correspond to the conditions of the rule (excluding WHERE conditions)
prb_recof {to be withdrawn}
This is a vector of numbers, recording relative "recency" information for the rule instance. There is one number for each condition, showing how recently the item matching it was added to the database, compared with other items in the same sublist of prb_database. This number is not as useful in this implementation as it was in the original implementation storing the whole database in a single list.

The number corresponding to a "complex condition", e.g. a WHERE, or ALL, or NOT or IMPLIES condition is 0. Otherwise it is a number >= 1 giving the relative "age" if the item in the sub-database containing it.
The vector will be empty ({}) if prb_recency is false.
(See information about prb_forevery, below.)

For example, if the conditions of the rule were

 [Father ?x ?y] [WHERE isfemale(y)] [Mother ?y ?z]

Then the prb_varsof field would contain the list [z y x], the prb_valsof field might contain [joe mary tom], the prb_foundof field
 [[Father tom mary] [Mother mary joe]]

and the prb_recof field might contain the vector

 {5 0 2}
 indicating that [Father tom mary] was the fifth most recent item added to the database and [Mother mary joe] the second. This information allows a variety of recency-comparing algorithms to be defined for ordering or selecting applicable rules.

WARNING: if a condition of type [ALL ?<variable>] is used to extend the conditions using patterns extracted from the database, as described below, this may also extend the prb_varsof and prb_valsof lists in instances of the rule, because the <variable> may be bound to a list containing variables, (e.g. extracted from the database). It may also extend popmatchvars for the rest of the rule. Thus if there are unintended repetitions of the variables in the rule and the patterns extracted from the database this can cause problems.

-- Notational convention [<data item>] [<pattern>] --------------------

In what follows the following conventions are used

 [<pattern>]
 is a list to be matched against database items. It may or may not contain pattern variables, some or all of which may already have been bound to values. For a summary of pattern syntax see HELP * MATCHES.

 [<data item>]
 is a list to be added to the database, or otherwise manipulated, possibly after instantiating variables, e.g.

 [isat fred kitchen]
 [isat ?person ?place]

 In the second form the variables are replaced by their current values before the list is dealt with. Before such a list can be added to the database, All variables must have been given values.

-- Types of rule conditions (simple and complex) ----------------------

Every rule has a (possibly empty) list of conditions determining when it is applicable.

Every condition is a list, which may represent a simple or a complex condition, as described below.

Applicability of all rules in the current ruleset is tested against the

current database (prb_database) on every cycle of prb_run, the top level procedure.
For a rule to be applicable all of its conditions must evaluate to true (though sub-conditions in a complex condition need not evaluate to true, e.g. in an OR condition). Also there are some pseudo-conditions which are always true and can be ignored, e.g. POP11 conditions.

On each cycle one or more of the rules whose conditions are true will be instantiated and the corresponding (instantiated) actions of the rule will be run.

There are several types of conditions, some of which are simple, i.e. they are merely patterns to be matched against the current database (held in the variable prb_database), while others invoke more complex mechanisms. There are also pseudo conditions which strictly speaking are not conditions to be checked but declarations or actions performed for their side effects, such as setting up a variable for use in subsequent conditions.

The permitted formats for conditions are as follows:

-- -- Simple conditions (patterns)

[<pattern>]
If the pattern does not begin with any of the keywords listed below as defining a complex condition, the condition as it stands is checked against items in the database. The condition is true if [<pattern>] matches something in prb_database, the working memory.

If any of the variables in [<pattern>] occur in previous conditions in the same rule, the match must be consistent with the variable bindings set up by matching the earlier conditions.

E.g. given these two conditions maternal grandfathers can be detected:
[father ?person1 ?person2]
[mother ?person2 ?person3]

The two occurrences of person2 must match the same item,

Any unbound pattern variables will become bound if the condition is satisfied. E.g. in the second condition ?person3 might become
bound to a grandchild of ?person1.

If person1 starts off unbound by previous conditions in the same rule, then by creating all the instances of the rule (achieved by setting prb_allrules true) we find all the people related as paternal grandfather/grandchild.

<pattern> may have any of the forms described in HELP MATCHES, including variables prefixed by "?" or "??" and variable restrictions, such as ?x:isnumber, ??list:3

As explained above, the wiser rule builder will refrain from incorporating rules with conditions that begin with variables,
as the matcher will then have to traverse the entire database, rather than looking only at the list of items indexed by the first element in the condition pattern.

Note that if the <pattern> matches several different items in prb_database, then different rule instances will be created corresponding to the different cases. Thus a rule with three conditions, each of which matches two database items could have eight rule instances created to deal with all the combinations of satisfied conditions.

This will not happen if prb_allrules (described below) is false. In that case, only the first combination of matches will be used to create a rule instance whose actions will be run.

-- . Some examples of simple conditions

```
[size ?object ?x]
[temperature ?t]
```

Patterns may contain embedded patterns, e.g.

```
[parts ?object == [?partnum == size 5 ==] == ]
```

which would access objects with parts, containing at least one part with attribute size 5, and will bind the variables object and partnum if such an object exists.

Restriction procedures may also be used, e.g. ?x:isinteger, or restrictions using user-defined procedures, as described in HELP * MATCHES.

E.g.

```
[age ?person ?num:isinteger]
```

This is generalised by the use of WHERE conditions, described below, which can be interspersed among ordinary conditions. These allow Pop-11 expressions to be evaluated in order to control testing for applicability. For example the following three conditions will be satisfied by two people, a, and b, such that b is twice as old as a, and a is older than min_age (where the latter is the value of a global variable).

```
[age ?a ?x1] [age ?b ?x2] [WHERE x2 == 2 * x1 and x1 > min_age]
```

In addition POP11 conditions and actions and "popval" or "apply" elements, explained below, allow Pop-11 expressions embedded in portions of a condition or action to be evaluated.

-- . NOTE on lexical scoping

The pattern variables in poprulebase (i.e. condition and action variables prefixed by "?" or "??" are all lexically scoped. Their scope is the current rule. Thus there will not be unwanted clashes with global variables using the same name defined outside the rule, as used to happen with early versions of this package and similar Pop-11 packages.

The price is that the words following "?" and "??" in patterns are replaced by identifier records when the rules are compiled. This can make tracing more messy, but to compensate the printing of identifier records is changed to show their current values, which can be helpful. See HELP *

LEXICAL.

 -- . "^" and "^^" will not work as expected in simple conditions.

Users who are familiar with the Pop-11 pattern matcher and its use in connection with the Pop-11 database, may be tempted to write conjunctive conditions sharing variables in this form:

```
 [father ?x ?y]
 [father ^y ?z]
```

e.g. in order to identify values of x and z such that x is the paternal grandfather of z. This will not work as expected. The Pop-11 syntax using "^" and "^^" to insert the value of a variable at run time is NOT supported in LIB POPRULEBASE conditions and actions, since these require too much re-creation of list structures when rules are run.

Instead the following syntax should be used

```
 [father ?x ?y]
 [father ?y ?z]
```

because the poprulebase interpreter will be able to tell that when the second condition is matched against database items the value assigned to y in the first match should be fixed for the second match. That is because all the conditions in a rule have to be true simultaneously with the same bindings for the same variables, as with Pop-11's allpresent procedure and the forevery construct.
 (See HELP * ALLPRESENT, * FOREVERY)

If you wish to insert in a condition or action list the value of some global variable, or some computed object, several techniques are available. The most general is the [LVARS ...] and [VARS ...] constructs, described below.

E.g. to create a condition that will find information about items whose speed is the same as the speed of the current object, where "current_object" is either a global variable or the value of a pattern variable set in an earlier condition:

```
 [LVARS [myspeed = speed_of(current_object)]]
 [speed item ?myspeed]
```

-- -- Using "^" or "^^" in conditions or actions at compile time

If "^" or "^^" is used in a condition or action, it may be followed by the name of a global (non-lexical) identifier (e.g. true, false) in which case the value of that identifier at compile time will be inserted into the list when the condition is created.
 -- -- Pseudo conditions

A rule can have pseudo conditions, which start with one of the words

 "DLOCAL" "LVARS" "VARS" "POP11" "DO"

These are described below.

-- . DLOCAL conditions

These can occur only at the beginning of a rule or a ruleset (or a rulefamily or a rulesystem).

They are explained fully in HELP RULESYSTEMS.

Their function is to set up an environment in which various global variables have specified values, e.g. to control tracing or the mode of operation of prb_run.

-- . [LVARS var1 var2 ....]

This can be used to declare variables for use in subsequent conditions, or actions.

The variables may thereafter be preceded by "?" or "??". Normally they will have to be given values in a POP11 condition. Sometimes the values are used in WHERE conditions also.

The formats can include initialisations, e.g.

 [LVARS v1 v2 [v3 = <exp>] v4 [[v5 v6] = <exp>] v7]

Then ?v1 ?v2 etc can be used in subsequent conditions or actions. If the value is a list, it is also possible to use the "??" prefix.

For examples of the use of LVARS see TEACH SIM_DEMO (if you have the full sim_agent kit installed.)

-- . [VARS var1 var2 ....]

This is like an [LVARS ...] condition except that it specifies that the variables should not be made lexically scoped, e.g. if for some reason you need to use a non-lexical global variable in a pattern. Again, initialisations are possible, e.g.

 [VARS v1 v2 [v3 = <exp>] v4 [[v5 v6] = <exp>] v7]

See HELP lexical

-- . [POP11 ...] conditions
[POP11 <pop-11 expression>]

 A condition of the above form is strictly equivalent to to a condition of the form [WHERE <pop-11 expression> ; true]
 (WHERE conditions are explained below.)

Thus all POP11 conditions will always succeed, and they can be used for tracing, or for setting variables for subsequent use in conditions or in actions, e.g. variables declared in [VARS ...] or [LVARS ...] conditions.
 All the comments regarding the use of variables in WHERE conditions are equally applicable to POP11 conditions. I.e. they should be pre-declared,

as the POP11 expressions are compiled when the rule is read in. (The use of "?" or "??" variables will prevent prior compilation, with a consequent loss of efficiency.)

The contents of the list, following the keyword "POP11" can contain ordinary Pop-11 instructions of any kind, e.g. for tracing and debugging purposes, showing what is happening during testing of conditions, or in order to set up values of variables to be used to constrain subsequent matches. To enable such variables to be accessible in subsequent conditions or in actions of the same rule, they should normally be previously declared using the [LVARS ...] format.

For example if the Pop-11 global variable "myself" is a pointer to some sort of record structure or objectclass structure, that has various "slots" and you wish to access the values of those slots in order to use them in conditions, then you can use LVARS to declare a variable and a POP11 condition to give it a value, so that it can be used in subsequent conditions.

E.g. suppose that one of the slots is called "agent_age". Then if you wished to have a condition that uses your age as a constraint in selecting another person, you could do this:

```
 [LVARS age]
 [POP11 agent_age(myself) -> age]
 [age ?person ?age]
```

This could also be expressed more compactly as

```
 [LVARS [age = agent_age(myself)]]
 [age ?person ?age]
```

These would be equivalent to, but more efficient than

```
 [age ?person ?age]
 [WHERE age = agent_age(myself)]
```

-- . [DO <action>]

A condition of the form [DO <action>] is a pseudo condition, which runs the action and then returns true.

-- -- Complex conditions

A complex condition starts with one of the key words described below, namely "NOT", "OR", "WHERE", "ALL", "NOT_EXISTS", "IMPLIES", "FILTER", etc. Complex conditions may perform arbitrarily complex tests.
 How these conditions work is defined by the recursive procedure prb_forevery and the sub-procedures that it invokes, in trying to find all possible ways of satisfying the conditions of a rule.

A full understanding of complex conditions requires users to be aware that typically when a condition C is being tested there is

 o an environment in which variables in preceding conditions have already been bound (i.e. pattern variables are treated as global).

 o a list REST of further conditions to be tested, whose testing will be

influenced by the way in which C is satisfied (i.e. how its variables are bound).

If C is not satisfied, then prb_forevery backs up to see if there are alternative ways of satisfying earlier conditions that will enable C to be satisfied. If not the current rule fails, and it will not be instantiated in this cycle. If C is satisfied, prb_forevery looks for all possible consistent ways of instantiating the remaining patterns in the list.

-- . [CUT] conditions.
The [CUT] pseudocondition, can be used to prevent the process of checking conditions against the database from going back to conditions preceding the [CUT] in order to try alternative instantiations of them.

The effect of this is to ensure that if all possible ways of satisfying the conditions to the right of the [CUT] have been tried then no further attempts will be made to check whether alternative matches for conditions to the left will be found. This is partly analogous to the use of the cut operator "!" in Prolog.

-- . OR conditions
[OR [<pattern1>] [<pattern2>] [<pattern3>] ...]
True if one of the patterns matches something in the database.

For each successful match of a sub-condition <patternN>, prb_forevery will try to match remaining conditions in REST until all remaining conditions match.

NB: at present "OR" can only be followed by a list of patterns to be matched against the database. I.e. OR is followed ONLY by simple conditions. Complex conditions are not allowed in an OR condition, e.g. conditions using "NOT" or "IMPLIES".

Because OR has this restriction, an OR condition actually functions as a simple condition in relation to various comments made below about restrictions on conditions. In particular, variables bound in an OR condition will be accessible in subsequent conditions and in ACTIONS.

-- . NOT conditions
[NOT <pattern>]
True if [<pattern>] does NOT match anything in the working memory. Initially unbound variables in the pattern may receive spurious values as a result of partial matches of <pattern>.

NOTE: the syntax is [NOT ?a on ?b] rather than [NOT [?a on ?b]]: the extra list brackets are not needed. If added they will cause the condition to fail if the database contains a ONE element list such as [[man on boat]]

Like "OR", this can include only a simple conditions, i.e. a pattern to be matched against the database, not complex condition. (NOT_EXISTS, defined below, can have arbitrary conditions, including complex conditions.)

Warning, although a NOT condition may contain variables, those variables should not be used (in the same rule) in subsequent conditions or in actions, as the values of the variables will be undefined if the NOT condition succeeds, and any values they have as a result of matching will

be implementation dependent and cannot be relied on.

-- . WHERE CONDITIONS
[WHERE <Pop-11 expression>]

This can be used to reject unwanted combinations of matches of preceding conditions, by imposing constraints in the form of a Pop-11 expression. Using WHERE conditions early in a conditionlist makes it possible to avoid wasted effort finding matches that are later going to be rejected.

The Pop-11 expression MUST produce a result, otherwise a stack underflow error will occur.

The condition is true if the expression evaluates to something non-false, otherwise the condition fails and the matching process will backtrack and try to find alternative ways of matching the preceding conditions with the database.

An example might be the following sequence of conditions.

```
[height ?person1 ?height1]
[height ?person2 ?height2]
[WHERE height1 > height2]
```

This will ensure that for subsequent conditions, and for actions of that rule, the variables person1, person2, height1 and height2, are constrained so that the first person has a greater height than the second.

If the WHERE condition uses an equality test it is often better to use a variable whose value is set in a POP11 condition, and incorporate it into a pattern, so that unwanted matches are detected earlier, as illustrated below in the section on POP11 conditions.
(This cannot be done so easily with an inequality.)

WHERE conditions will usually contain variables that have been bound in a previous pattern, as in the previous example. However, the variables in a WHERE condition should not be preceded by "?", since the expression will be compiled before the rule is created, and is treated as normal Pop-11 code. In Pop-11 forms such as

```
?height1 > ?height2
```

are illegal and will produce an error. For the present, as an experiment, the package allows variables preceded by "?" or "??" and in that case the expression will not be compiled when the rule is read in. Instead it is instantiated and compiled when checking occurs. That is far less efficient, and should not normally be used except in very special circumstances (e.g. where there are macros that are redefined at run time).

Note that in a WHERE condition, variables are used with whatever values they have got from previous conditions or the environment. However, words that are not variables but refer to themselves need to be explicitly quoted,
e.g.

```
[man isat ?place] [?thing isat ?place] [WHERE thing /= "man"]
```

I.e. [WHERE thing /= man] would not work as expected in the last condition. In simple conditions words are quoted by default, whereas in a WHERE condition they are treated as part of the Pop-11 text and are therefore unquoted by default.

If a WHERE condition is to create a list and that list needs to include the value of a variable set in an earlier condition, then use "^" and "^^" as if it were ordinary Pop-11 code.
 (See TEACH * ARROW).

WHERE conditions do not have to come at the end of a condition sequence. E.g. the above conditions might be followed by

 [owner ?thing ?person1]

if the rule is restricted to things that have owners, and subsequent conditions or actions need to be able to access the owner.

 If the Pop-11 expression in a WHERE condition causes an error, e.g. because not all the variables have appropriate kinds of values, then this is equivalent to the WHERE expression being false. (The errors are trapped. Such trapping is turned off by prb_debugging)

Note that if you use variables in a WHERE condition these should be declared prior to the rule, since otherwise the system will attempt to autoload them and that can slow compilation down and produce unwanted side effects, or at the very least "DECLARING VARIABLE" messages.

It is good practice to declare all variables used in conditions, as these are essentially global variables.

-- . INDATA conditions [INDATA <database> <condition>]

The condition keyword "INDATA" makes it possible for a rule to check one or more conditions against a database other than the current default database.

This could be useful for a type of application of the Sim_agent toolkit in which one agent can read another's database. For example if fred is another agent, then a rule in the rulesystem of another agent could have these two in its conditions:

 [LVARS [fred_data = sim_data(fred)]]
 [INDATA ?fred_data [hunger level ?val]]

Then this will evaluate to true if the agent fred has in its database a list something like
 [hunger level high]

In that case the variable value will obtain the word "high" as its value.
 The [LVARS ...] declaration could come at the beginning of a whole ruleset if several of the rules need to be able to access fred's database.

The use of several databases could lead to considerable time-saving if it splits up long lists of data items into short lists.

-- . NOT_EXISTS and IMPLIES conditions

A particularly powerful type of condition allows a rule to be fired only if some generalisation is true. For example if you want a rule to fire if all known adult males are teachers you could use a condition of the following form:

```
 [NOT_EXISTS [male ?x] [adult ?x] [NOT teacher ?x]]
```

Because this can be hard for non-logicians to interpret, the following format is also accepted for the special case of a NOT_EXISTS condition that has only one NOT condition.

```
 [IMPLIES [ [male ?x] [adult ?x] ] [teacher ?x]]
```

(NB earlier versions of this help file had a spurious NOT in the last condition, but nobody complained.)

This uses the logical equivalence:
  (C1 & C2) -> C3 if and only if not(C1 & C2 & not C3)

Note 1: Notice that "IMPLIES" is followed by only TWO components, the first being a LIST of conditions (i.e. the antecedents of the implication), and the last a single condition, the consequent.

Note 2: NOT_EXISTS can have any number of conditions. If there is only one condition and that is simply a pattern, then it is equivalent to a "NOT" condition, though slightly less efficient.

Unlike "NOT", "NOT_EXISTS" can be followed by any type of condition, simple or complex, whereas NOT can be used only with simple conditions.

E.g. the following is legal

```
 [NOT_EXISTS [OR [<pattern1>] [<pattern2>] ...] ]
```

whereas the corresponding "NOT" condition would not work.

WARNING: although the patterns in a complex NOT_EXISTS or IMPLIES condition may contain variables, and these variables may be repeated within the same complex condition, the same variables should NOT be used (in the same rule) outside that conditions, as their values will be undefined. E.g. do not do things like

```
 [IMPLIES [human ?x] [mortal ?x]]
 [age ?x ?age]
```

as the value of x in the last condition will have no clear meaning.

-- . ALL conditions (for meta-rules)

[ALL ? <variable1>]
 This form of condition is provided to allow a list of patterns to be dug out of the database and then checked as if they were part of the list of conditions of the rule.

Suppose, for example, that there is a goal in the database of the form

 [goal [BlockA ison ?block] [?block ison =]]]

Then a rule that is to check whether that goal is satisfied might have the following two conditions

 [goal ?goallist]
 [ALL ?goallist]

The first condition will dig out the goal. The second will check whether it is satisfied, and if so bind the list of database items to the variable instances.

Both the variable ?goallist and variables that occur in it can be used in actions of the rule. An example of this mechanism is shown in
TEACH * PRBRIVER
 This is another context that depends on the fact that the variables in conditions are all essentially global variables.

Note that if an action is to use goallist after it has been matched against a database item containing variables, and some of the variables have been bound, then the action cannot simply use the variable, e.g.

 [??goallist]

cannot be used to add the instantiated list items to the database. Instead the variables in goallist will have to be instantiated.
This could be done in a pop11 action, e.g.

 [POP11 prb_add(prb_instance(goallist))]

The action type "ADDINSTANCE" is provided to cope with this. So
the following would work:

 [ADDINSTANCE ??goallist]

This uses the procedure prb_addinstance, defined in LIB * POPRULEBASE

See also TESTADDINSTANCE

-- . FILTER conditions

FILTER conditions, in combination with MAP and SELECT actions, allow non-rulebased systems (e.g. neural nets or other "sub-symbolic" mechanisms) to be smoothly integrated with a rulebased system. These are described in detail in HELP * PRB_FILTER but are summarised here for convenience.

 [FILTER BFP C1 C2 ... Cn]
 Where BFP is a boolean filter procedure, and C1...Cn are
 simple conditions.

 [FILTER VFP -> var C1 C2 ... Cn]
 Where VFP is a "vector filter procedure", and var is a variable holding its result, and the Ci are conditions.

The MAP and SELECT actions for use with FILTER conditions are also

described in HELP PRB_FILTER

NB this form of FILTER condition can be clumsy to use if you wish to run a neural net on all possible combinations of values of certain variables. A more flexible form is under consideration.

-- . Condition handles [->> var]

[->> var]
 This form of condition is a pseudo condition, which must be preceded by a simple condition. The [->> ...] condition always evaluates to true, and has the side effect of assigning the item in the database that matched the previous condition to the variable following the arrow.
  This allows a pointer to a database item matching a condition to be used in subsequent conditions or in actions, without having to reconstruct the item from its parts.

This was suggested by Darryl Davis and Riccardo Poli.

It is then possible for subsequent conditions and subsequent actions in the same rule to use ?var or ??var to refer to that database item or its contents.

For example the variable can be used with DEL, REPLACE or MODIFY actions, or in WHERE conditions or POP11 conditions or actions.

-- Actions --

Each rule has a list of actions. On every cycle, prb_run selects one or more rules that are runnable and then executes each of their actions in turn.

Actions, like conditions, may be simple or complex. Complex actions are lists which start with a known keyword (e.g. READ, SAY, MENU, DEL, MODIFY, POP11 etc -- all defined below). Simple actions are implicit ADD actions, i.e. they are lists to be added to the database.
 Some actions are pseudo actions, as they modify the environment for actions which follow, e.g. [LVARS ...] and [VARS ...]

The notation used below corresponds to the notational convention introduced above.

-- . Simple actions, implicit add: [<data item>]

The actual list, [<data item>] is added to the database.

A simple action is simply a list that does not start with any of the action keywords listed below. After all variables in the list have been instantiated the list will be added to the database, e.g.

 [?item isat ?place]

could be a simple action to add three items (although the Pop-11 list contains five items "?", "item", "isat", "?", "place". The current values of item and place, usually derived from the conditions of the rule, will be used to instantiate the action before the list is added to the

database.

If one of the variables has not been used as a variable in a condition, but the value has been computed in a POP11 action, it can be predeclared as a pattern variable in an LVARS pseudo action (described below), so that the above syntax using a pattern variable works:

```
 [LVARS place]
 [POP11 .... -> place; ....]
 [?item isat ?place]
```

[LVARS ...] and [POP11 ...] actions are explained below.

-- . prb_auto_add

The global variable prb_auto_add, which defaults to true, allows these implicit add actions. If it is false, then an explicit add action of the form [ADD ...] described below, must be used. Otherwise an unrecognised action type mishap occurs at run time.

-- . Explicit add: [ADD <data item>]

This is exactly equivalent to [<data item>] as defined above, except that its behaviour is independent of the value of prb_auto_add.

It uses more space in rules and rule instances, but the reduced efficiency is compensated for by the fact that mistakes in typing
action keywords are more likely to be detected.

NOTE:
The format for this was previously wrongly specified as follows.
 [ADD [<data item>]]

This help file was corrected 26 Jun 1998
 -- . Non-duplicate add: [TESTADD <data item>]
-- . Non-duplicate add: [TESTADDINSTANCE <data item>]

Like ADD, or ADDINSTANCE, respectively. The second is required when the <data item> is a pattern (e.g. something found in an ALL condition) and what is to be added is an instance of the pattern.

Unlike ADD and ADDINSTANCE, these two first check whether anything already in the database matches [<data item>] and if so the ADD or ADDINSTANCE is not performed. If nothing matches then the item is added. For large databases this can be much slower than the previous two kinds but it may sometimes be essential to prevent duplication of database items.

-- Built-in complex action types ----------------------------------

Complex actions corresponding to initial keywords may either be of a built-in type or a user-defined type, as explained below. All actions are instantiated before being executed, as described in the section on instantiation, below.
 The main form of instantiation is replacement of variables, indicated by "?" and "??" by the values bound to them when conditions are checked.

Additional variables for use in actions can be set up using the "VARS" or

"LVARS" pseudo actions described below.

The built-in complex action types are as follows:

-- . [ADDALL [<data item>] [<data item>] ...]

All the items are added to the database, in the order given. This means that the last one will be the most recent, which may be important if "recency" information is used for selecting rules (See prb_recency and prb_recof).

-- . [ADDINSTANCE <pattern>]

This performs the action
 prb_add(prb_instance(<pattern>))

-- . [ADDIF [<pattern>] [<data item>] [<data item>] ...]
-- . [ADDUNLESS [<pattern>] [<data item>] [<data item>] ...]

These can be used to put a "guard" on the addition of items to the database. The (instantiated) pattern is checked against the database (using prb_present) and if a match is found, in the first case, or not found, in the second case, the <data items> are added to the database.

-- . [SAY <message>]
[<message>] is printed out. It can be a mixture of "canned" strings, variables preceded by "?" or "??", [popval...] or [apply...] elements, or embedded lists. The variables that have been bound will be replaced before printing.

-- . [SAYIF <keyword> <message>]

This is for conditional tracing. If <keyword> is a member of the the list prb_sayif_trace, then [<message>] is printed out.

It can take the same forms as in "SAY" actions.

-- . [EXPLAIN <message>]
This is exactly like SAY actions except that if prb_explain_trace is false then it is ignored. For example this can be used to suppress verbosity during development.

-- . INDATA actions [INDATA <database> <action>]

The action keyword "INDATA" (which is also described above as a condition keyword) makes it possible for a rule to perform an operation (e.g. adding or removing a list) in a database other than the current default database. This could be useful for a type of application of the Sim_agent toolkit in which one agent can modify another's database. For example if fred is another agent, then a rule in the rulesystem of another agent could have these in its actions:

```
 RULE increase_hunger
 ;;; find fred's hunger level
 [LVARS [fred_data = sim_data(fred)]]
```

```
 [INDATA ?fred_data [hunger level ?oldval]]
 ==>
 ;;; increment the level
 [LVARS [val = oldval + 2]]
 ;;; delete the old entry
 [INDATA ?fred_data [NOT hunger level ?oldval]]
 ;;; add a new one
 [INDATA ?fred_data [hunger level ?val]]
```

Then this rule will cause the information about hunger level in fred's database to be removed and replaced with a new entry giving more up to date information.

It would also be possible to use a single [MODIFY ...] action (described below) with INDATA instead of the two actions to delete and insert database items.

The [LVARS ...] declaration could come at the beginning of a whole ruleset if many of the rules need to be able to access fred's database.

The use of several databases could lead to considerable time-saving if it splits up long lists of data items into short lists, thereby reducing database search times.

-- . [STOP <message>]
This is exactly like a SAY action, except that after the <message> is printed out the production system run is halted.

-- . [STOPIF <item> <message>]

This is explained in HELP RULESYSTEMS/'[STOPIF'.

Basically, if <item> is false this does nothing. If it is true, it behaves like [STOP <message>]

-- . [QUIT <message>]
-- . [QUITIF <item> <message>]

These two are explained in
 HELP RULESYSTEMS/'[QUIT'.
 HELP RULESYSTEMS/'[QUITIF'

They can be used to terminate the current rule's actions without terminating the current run of the rule interpreter.

-- . [NOT <pattern>]
All items matching <pattern> are removed from the working memory.
No extra list brackets are required for <pattern>. E.g.

```
 [NOT boat at left]
 will delete the item [boat at left] from the database.
```

-- . [LVARS var1 var2 ...]
-- . [VARS var1 var2 ...]

These forms of action can declare additional variables for use in

subsequent actions. Normally they are needed in connection with POP11 actions, as explained below. The [LVARS ...] form should normally be used as it will make the variables lexically scoped. Use the [VARS...] form when you need to access a non-lexical global variable.

VARS and LVARS actions, like VARS and LVARS conditions, can also include initialisation of the variables declared.

E.g.
```
 [LVARS v1 v2 [v3 = <exp>] v4 [[v5 v6] = <exp>] v7]
```

```
-- . [POP11 <procedure>]
-- . [POP11 <procedure name>]
-- . [POP11 <pop11 instructions>]
```

In the first two cases the procedure is run. In the third case the instructions are obeyed. Note that the instructions are compiled to a procedure at the time the rule is read in, unless they make use of "?" or "??" variables. In this respect a POP11 action is like a POP11 condition.
 It is possible to use this form to handle certain conditional actions efficiently, instead of having separate rules for each case. An example might be:

```
 RULE temperature
 [temperature ?temp]
 ==>
 [POP11
 prb_add(
 if temp == 98 then [temperature normal]
 elseif temp < 98 then [temperature low]
 else [temperature high]
 endif)
 ]
```

This could be expressed more clearly as three separate rules using WHERE conditions, but in some cases the loss in efficiency might be important.

Another typical use of a POP11 action is to set up variables. For example if obj is a variable that has been bound in the conditions of the rule, you can do things like this in the actions:

```
 [LVARS obj_name]
 [POP11 name_of(obj) -> obj_name]
 [ ... ?obj_name ....]
```

Where the third action asserts something in the database using the value of the slot name_of(obj). Similarly POP11 actions can be used to change values of data-structures other than the prb database.

Note that the two procedures
```
 prb_eval(<action>)
```
and
```
 prb_eval_list(<action list>)
```

are available for use in the body of a POP11 action. The <actions> can be

any items of the sort that can occur as actions of a rule, for example

 [POP11 if ... then prb_eval([DEL 2]) endif]

would, under appropriate conditions, be equivalent to the action

 [DEL 2]

Similarly

 [POP11 if .... then prb_eval_list([.....]) ....endif]

The only use for these procedures arises when the POP11 code calls the actions indirectly, e.g. inside a conditional.

-- . [PUSH <item> <word>]
This assumes that there is in the database a list of the form
 [<word> ....]

This is replaced with

 [<word> <item> ...]

If prb_copy_modify is false, the old list-links are used, otherwise the old item is removed and a new one constructed. However, in either case the tail of the original list (represented by ....) is re-used.

-- . [POP <word>] This assumes that there is in the database a list of the form
 [<word> <item> ....]

This is replaced with

 [<word> ...]

I.e. the first item after the stack name (<word>) is removed. If prb_copy_modify is false, the old list-links are used, otherwise the old database item is removed and a new one constructed. However, in either case the tail of the original list (represented by ....) is re-used.

NB Don't confuse POP actions and POP11 actions.

-- . [DEL <integer> <integer> ...]
-- . [DEL ?var1 ?var2 ...]

For each <integer> N, delete from the working memory the item that matched the N'th simple condition of the rule (i.e. excluding WHERE conditions, NOT conditions ALL conditions and other complex conditions).

Where a "?" variable is used, it should have as value an item from the database, which will then be removed. E.g. it might have been set up by a condition of the form [->> var1]

The second form is the preferred mode of use of "DEL" actions as it is

less liable to lead to bugs were the conditions in a rule are changed and the <integer> is not altered, or bugs due to miscounting.

-- . . WARNING count only SIMPLE conditions for <integer>
A common cause of errors is to count complex conditions in selecting the integers in DEL, REPLACE, MODIFY or RMODIFY actions (see below). For this reason it is probably best to use the variable format.

-- . [REPLACE <integer> [<data item>]]
-- . [REPLACE ?var [<data item>]]

This is equivalent to the two actions

 [DEL ...] ;;; using the integer or variable.
 <data item>

but may be a little clearer to read in some contexts, as well as being more compact.

Note the warning above about how to interpret the integer, and the note about the second form being preferable.

-- . [REPLACE [<pattern>] [<data item>]]

This is equivalent to two actions
 [NOT <pattern>]
 [<data item>]

Except that only one instance of <pattern>, the first found, will be removed.
 Warning: don't use variables in <data item> that you expect to be bound in the <pattern>. If instantiated variables are to be used to specify the <data item> then they must either be bound in the conditions of the rule, or declared earlier in a [LVARS ...] pseudo action. They can be initialised in the LVARS form, or in a subsequent POP11 action.

E.g.

 [LVARS [person =

-- . [MODIFY <integer> <key> <value> <key> <value> ...]
-- . [MODIFY ?var <key> <value> <key> <value> ...]

If the integer is N, this is equivalent to a command to remove the database item that matched the N'th SIMPLE condition, and replace it with a new copy in which the item following the occurrence of <key> is replaced by an occurrence of <value>, for each <key> <value> pair in the action.

Similarly, if the value of "var" has been set by a [->> var] condition, the corresponding database item will be used.

As with "DEL", the second form is the preferred mode of use, as it is less liable to lead to bugs were the conditions in a rule are changed and the <integer> is not altered, or bugs due to miscounting.

EXAMPLE:
If the first condition of a rule is [monkey == holds ?x ==] then the

action [MODIFY 1 holds bananas] would mean that whatever matched x would be replaced by the word "bananas". This is useful when it is necessary to modify part of a database item whose complete structure is not known.

Note that as far as the integer is concerned, only SIMPLE conditions are counted. E.g. if the first condition of the rule is of the form [NOT ...] and the second condition is [monkey == holds ?x ==], then the [MODIFY 1 ..] action would refer not to the [NOT ...] condition, since it is a complex condition, but to the [monkey ...] condition, which is simple (a pattern). See comment on DEL actions above.

WARNING: do not use two MODIFY commands with the same <integer> or variable in the actions of the same rule. E.g. don't do:

[MODIFY 2 age 20]
[MODIFY 2 wife mary]
The second modify action will not find the original database item to modify. Instead do

[MODIFY 2 age 20 wife mary]

This is more efficient in any case as it requires the modified item to be searched for in the database only once.

WARNING: if one of the <key>s happens to be identical with one of the <value>s in the database item, then the item following the <value> may get replaced, causing obscure bugs. Checking for this would slow things down and require restrictions on formats used.

-- . [MODIFY <pattern> <key> <value> <key> <value> ....]
As above, except that the modification is done to everything that matches the <pattern>.

Warning: do not expect variables bound in the pattern to provide values for other variables in the action, unless they are declared in VARS. See the comment on REPLACE actions.

-- . [RMODIFY <integer> <key> <value> <key> <value> ...]
-- . [RMODIFY ?var <key> <value> <key> <value> ...]
-- . [RMODIFY <pattern> <key> <value> <key> <value> ....]
Recursive Modify: These are like the [MODIFY ...] action forms except that the replacements are done in embedded lists also. Beware: ALL the embedded lists must have even numbers of items and must be of the form [<key> <value> <key> <value> .... <key> <value>]

Also any embedded lists which are to be modified must occur in a <value> position, not a <key> position. For example, if these conditions occur in a rule

[goal ?id state [actions [do [goto ?location fetch ?thing] ....]]] [->> Cond]

Then this action will work:

[RMODIFY ?Cond fetch ?newthing]

If prb_copy_modify is true then the complete data item matching the original pattern is replaced with a new one. If the variable is false, then the existing database item has one of its components replaced.

-- . [NULL .....]
If an action starts with "NULL" nothing is done, apart from instantiating it. This instantiation means that it can include variables that will be bound and [popval ...] items or [apply ...] items that will invoke Pop-11 procedures, e.g. to display the database or do some book-keeping. E.g.

```
 [NULL [popval prb_database ==>]]
```
or
```
 [NULL [$$ prb_database ==>]]
```

will simply print out the whole database.

A more efficient equivalent action form would be:

```
 [POP11 prb_database ==>]
```

See [POP11 ...] actions.

-- . [RULE TYPE <ruletype> <name> [<conditions>] [<actions>]]
or
-- . [RULE <name> [<conditions>] [<actions>]]

Note, an integer weight can occur after <name>

This creates a new rule (added to the end of the list of name <ruletype>), or replaces an old rule if it already exists in the list. If "TYPE <ruletype>" is not specified then it defaults to "prb_rules". If the ruletype is specified, then the corresponding ruleset must already exist, though it could be an empty list.

WARNING: if the new rule is to include any variables preceded by "?" or "??" then make sure their names are different from any variables occurring as rule variables in the rule containing the [RULE ...] action.
 Here's an example action

```
 [RULE
 TYPE elizarules ;;; ruleset name
 myrule ;;; new rule name
 [[sentence I ??words you] [user friendly]] ;;; conditions  [[SAY Perhaps we ??words each other]] ;;; actions
 ]
```

The following detects any item containing an even number in the database, removes it, and creates a new rule that will abort the process if another even number turns up.

```
 RULE rule check_even
 [== ?x:isinteger ==]
 [WHERE x mod 2 == 0]

 ==>
```

```
 [DEL 1] ;;; delete it

 [RULE
 [popval gensym("rule") ] ;;; new rule name  [[== ?y:isinteger ==] [WHERE
y mod 2 == 0]] ;;; conditions  [[STOP Found even number ?y]]] ;;; actions
```

See also prb_new_rule

-- . SAVE and RESTORE actions (for rules or data)

It is possible to save or restore either the working memory, or the ruleset using these actions. The formats available are as follows:

```
 [SAVE RULES <name>]
 Equivalent to: prb_rules -> valof(<name>)

 [SAVE DATA <name>]
 Equivalent to: prb_database -> valof(<name>)

 [RESTORE RULES <name>]
 Equivalent to: valof(<name>) -> prb_rules

 [RESTORE DATA <name>]
 Equivalent to: valof(<name>) -> prb_database
```

These actions can be combined, as follows:

```
 [SAVE DATA <name1> RULES <name2>]

 [RESTORE DATA <name1> RULES <name2>]
```

If the ruleset is changed using [RESTORE RULES ...] any remaining rules in the currently applicable list are run.

The above actions are not suitable for use with the SIM_AGENT toolkit if many agents are used. That is because the use of global variables can cause agents to interfere with one another, and also between runs of an agent the interpreter needs to know whether a ruleset transferred control to another ruleset. Consequently a different mechanism is provided for sim_agent applications, using rulefamilies described in HELP * RULESYSTEMS

-- . Action types provided by LIB PRB_EXTRA

LIB * PRB_EXTRA provides additional facilities for temporarily changing the ruleset or database by pushing or popping them.

These new actions are described fully in HELP * PRB_EXTRA

```
-- . . [PUSHRULES <ruleset>]
-- . . [POPRULES]
-- . . [POPRULES <ruleset>]
-- . . [PUSHDATA <database>]
-- . . [PUSHDATA [<patternlist>] <database>]
```

-- . . [POPDATA]
-- . . [POPDATA [<patternlist>]]
-- . . [POPDATA <database>]
-- . . [POPDATA [<patternlist>] <database>]

Additional facilities described in the help file are
 prb_clear_stacks()
 prb_current_data_stack()
 prb_current_rule_stack()

-- . Action types for use with FILTER conditions

-- . . [SELECT ?var A1 A2 ... An]
-- . . [MAP ?var MP A1 A2 ... An]

These two are described in HELP * PRB_FILTER. They enable the output of a vector filter procedure in filter condition to control the selection of actions in an action list.

-- . [DOALL A1 A2 ... An]

This is used for embedded actions that all need to be obeyed as if they were top level actions with all the options described here. The procedure prb_eval_list, is given the list [A1 A2 ... An]

-- . [FAIL <message>] [WITHDRAWN JULY 1995]

-- Manipulating dependencies [DADD ...] and [DDEL ...]

Two action formats are provided for manipulating dependencies between facts stored in a database.

[DADD <datum> <j1> <j2> <j3> ...]

 where j1, j2, ... are database items (or patterns which can match database items), adds new information saying that j1,j2,... jointly suffice to justify datum. There may be alternative sufficient justifications for the same datum.

[DDEL <datum>]
 This deletes <datum> and also anything which depends on it and has no other remaining justification list.

For full details of these two action forms, and examples see

 HELP * prb_DADD

-- Actions for tracing and interacting with the user

-- . [PAUSE]
 This action will pause till the user presses the RETURN key, unless the variable prb_pausing has the value false. It is equivalent to a READ action of the form: [READ '' [==] []]

This means that interactive commands can be given during the pause,

as explained in the section on Interactive commands below.

-- . [READ <question> <??constraint??> <data item> <??explanation??>]

The <constraint> is optional and may be omitted. The <explanation> is also optional and may be omitted. The forms of the elements of READ actions are described below.

When a READ action is executed, the <question> is printed out, and the user types something in which is tested against the <constraint> (if provided).

<question> may be a string or list or any other object that can be printed out.

<data item> should be a list. It may contain the word "ANSWER", in which case that will be replaced by whatever the user types in, and the resulting <data item> will be added to the database, unless it is the empty list.
 The <constraint>, if present, should take one of the forms below, and if omitted it defaults to [==], which will match anything

The <explanation> should be a vector containing strings, words or lists, which, if requested, will be printed out suitably instantiated. Strings are printed without alteration. A word is replaced by its current value. A list will be evaluated using prb_value, which will replace variables of the form ?x or ??x with their values, and lists of the form [popval <expression>] as explained below.

A READ action ending with an explanation may have the form:

```
 [READ
 ['Does' ?patient 'feel ill, well or soso?'] ;;; question
 [OR ill well soso] ;;; possible answers
 [patient feels ANSWER] ;;; to go in database
 {'Because how' patient 'feels is relevant to the diagnosis'}]
```

A more detailed account of READ actions follows.

-- . . READ actions in detail

The READ action of the above form does the following: (1) <question> is printed out,
 (2) A line is read in (using -readline-), but with an opportunity for the user to ask questions as explained in the section
 on interactive commands below.
 (3) The answer is matched against <constraint> and if it matches then (4) It is inserted in place of every occurrence of "ANSWER" in
 <data item> and the latter is added to the database, unless
 it is empty.

 If the match fails at (3) the process restarts from (1).

An empty action as in [READ <message> []] is useful for ensuring that the program pauses and gives the user the chance to invoke the interactive commands described below.

-- . . READ action constraints

If the constraint in a READ action is of the form [:<word>] then that implies that only one item should be typed in, and <word> is the name of a procedure that must be applied to that item and return a non-false result. E.g. in order to insist on an integer, use the constraint [:isinteger]
If the constraint is of the form [OR <item1> <item2> ...] then the user's input must be a single item identical with one of <item1> <item2> etc.

If the constraint is of the form [LOR <item1> <item2> ...] then the user's input must be a list identical with one of <item1> <item2>
etc.

So the following two constraints are equivalent
[OR a b c] [LOR [a] [b] [c]]

However the second format allows [] to be one of the options, so that an empty reply is acceptable where a default can be used.

If the user is to type in several items separated by spaces, and these are to be put in a list satisfying some condition, then the constraint could be of the form

[?? <variable> : <word>]

where <word> is the name of a procedure, which will be applied to the list and should return true or false. E.g. if the procedure -isnumberlist- has been defined so that it checks whether its argument is a list of numbers or not and returns -true- or -false-, then the constraint might be something like:
[??numbers:isnumberlist]

In that case the value of the variable "numbers" can be used in subsequent actions.

Instead of an answer to the question the user may type "show", "why", or one of the other options specified in the section on interactive commands below.

The default READ facility makes use of -readline- so all answers have to be typed on one line. (See HELP * READLINE)

-- . [MENU <menu> <action list> <??explanation??>]

This is analogous to a READ action, except that it can offer users the option of a menu of answers to choose from, so that the user need simply reply by typing in a letter or number instead of having to type in the whole thing.

As with READ actions, the <explanation> is also optional and may be omitted. The format for explanations is as described above for READ actions.

When a MENU action is executed, the <menu> is printed out, and the user types something in which is tested to make sure it is one of the options provided in the menu. If it is, the appropriate action from the

<action list> is executed. The <menu> and <action list> must have related structures as follows.

The <menu> is a vector of two or three elements of the following form

```
 {[<message>] [<options>] [<mappings>]}
```

Where the last item is optional: it is useful only in the case where the action list is of the first form indicated below.

The <action list> must be either a list containing a single action (in the form of a list, possibly including the word "ANSWER") or else a list mapping options to actions, as explained below.

Here is a simple example of a menu with message list, options list, and mappings list:

```
 [MENU
 {['Does it produce milk?']
 [[1 'yes it does'] [2 'no it does not'][3 'dont know']]
 [[1] [milk yes] [2] [milk no] [3] [milk unknown]]
 }
 ]
```

-- . . Structure of the <menu> vector

In more detail the structure of a <menu> vector is this:

The first element of the vector, [<message>] is a list of words or strings that can be printed to pose a question, prior to printing out the menu of possible answers. If the words are preceded by "?" or "??" they are assumed to be variables and their values are substituted instead. Any embedded "popval" elements or "apply" elements are evaluated as described above.

A possible example would be

```
 ['How does' ?patient 'feel today?']
```

-- . . Structure of the menu [<options>] list

The second element of the <menu vector> [<options>] is a list of lists, one list for each option. Each list starts with a letter or number, to be typed to select that option and continues with information to be printed out to describe the option. An example of the options list might be:

```
 [[1 very ill] [2 very well] [3 soso] [4 dont know]]
```

-- . . Structure of the menu [<mappings>] list

The third element of the <menu> vector, [<mappings>] is a list showing what value should be assigned to the variable ANSWER corresponding to each selection made. The list is of the form

```
  [<key> <value> <key> <value> <key> <value> ....]
```

where each <key> must either be an ATOMIC object, i.e. a word or a number that can be tested using "==" or a list of such items.
(See HELP * EQUAL for information on the difference between "==" and "=" for testing equality.)

For example the list of mappings might be:

```
 [ 1 ill 2 well [3 4] [not known]]
```

So if the user typed 1 the word "ill" would be assigned to ANSWER, if the user typed 2 the word "well" whereas if the user typed either 3 or 4 the list [not known] would be used. If this list of mappings were not available, the value of ANSWER would be whichever number the user selected.

Each <value> may be any arbitrary item or a list of items, unless the second sort of action list described below, is used, namely a list which maps menu keys to actions. In that case each <value> in the mappings list must also be a word or number which can be tested using "==", not a list, or string, or other data-structure. So the [not known] <value> would not work. This is explained with an example, below.

-- . . The menu <action list>

When an appropriate menu option has been selected by the user, the program performs the appropriate action chosen from the action list.

The <action list> has one of the following formats

-- . . . Single action list
The list contains one item, namely a list, giving an action to be performed, with the value of ANSWER replacing occurrences of "ANSWER":

```
 [[<action>]]
```

```
 e.g. [[Patient needs ANSWER]]
```

or

-- . . . Keyed action list
The list contains a number of <key> [action] pairs in this format:

```
 [<key> [<action>] <key> [<action>]...]
```

where each <key> is an item, or a list of alternative items, that the user may have typed in, or may have been derived from what the user typed in, on the basis of [<mappings>]. The key is then used to select the action, which is carried out. An example of a keyed action list for a menu would be:

```
 [
 ill [?patient needs treatment]  [well unknown] [?patient needs tests]
 ]
```

-- . . . Types of menu actions
In the single action format, with a list containing a single action, the <action> may, as with READ actions, include the word "ANSWER", which will then be replaced either by what the user typed in, or by the item related to what the user typed in, according to the <mappings> list, described above.

Each menu <action> in either the single action format or the keyed action format can be any action type that can occur in a rule, including a further MENU action.

WARNING: If the mappings list is provided as well as a keyed action list, then each <value> in the mappings list MUST be an ATOMIC object, i.e. a word or number, which can be compared using "==". I.e. it must not be list or string or vector. Then each <key> in the <actions> list should either be one of those <values> or a list of those <values>.

For example, the following violates that constraint and would not work as expected:

```
 Options list:  [[1 I feel very ill] [2 I am well] [3 soso] [4 dont know]]

 Mappings list:
 [1 [very ill] 2 [very well] [3 4] unknown]

 Actions list:
 [
 [very ill] [?patient needs treatment]
 [[very well] unknown] [?patient needs tests]
 ]
```

If the user typed in "1" the list [very ill] would be provided as the value from the mappings list, but this would not be matched against the first item in the actions list, rather it would be compared with the word "very" and with the word "ill", and the test would fail. If the value 2 were typed in by the user, then the mappings list would produce the value [very well] and this would not be recognised as a member
of the second list of keys in the actions list, because "lmember" is used, for speed, and it uses the test "==" not "=".

So for communication between bits of a MENU action use words or numbers, not complex objects like lists or strings or vectors. This should not impose any restriction, since these values are for internal use only.

-- . . Example of a MENU action

The following example assembles the items discussed above:

```
 [MENU
 {
 ;;;<message list>
 ['How does' ?patient 'feel today?']
 ;;; <options list>
 [[1 very ill] [2 very well] [3 soso] [4 dont know]]
 ;;; <mappings list>
 [ 1 ill 2 well [3 4] unknown]
 }
```

```
 ;;; action list allowing keys "ill" "well" "unknown"
 [
 ;;; single key
 ill [?patient needs treatment]

 ;;; two possible keys associated with the same action
 [well unknown] [?patient needs tests]
 ]
 ;;; explanation
 {'Because how' patient 'feels is relevant to the diagnosis'}]
```

An equivalent MENU action, not using a <mappings list> would be:

```
 [MENU
 {
 ['How does' ?patient 'feel today?']
 [[1 very ill] [2 very well] [3 soso] [4 dont know]]  ;;; no <mappings
list>
 }

 ;;; action list using the numbers directly as keys
 [
 1 [?patient needs treatment]
 [2 3 4] [?patient needs tests]
 ]

 {'Because how' patient 'feels is relevant to the diagnosis'}]
```

Yet another equivalent would be

```
 [MENU
 {
 ['How does' ?patient 'feel today?']
 [[1 very ill] [2 very well] [3 soso] [4 dont know]]
 [1 treatment [2 3 4] tests]
 }

 ;;; Single action format, using "ANSWER"
 [
 [?patient needs ANSWER]
 ]

 {'Because how' patient 'feels is relevant to the diagnosis'}]
```

As the last two examples show, there is usually no point having a <mappings> list in the menu vector unless it is required to associate the user's choice with some other value for ANSWER to be inserted in some data base item or action.

-- . . Printing out menus: prb_print_menu(message, options)
 The printing of a menu is done via the user-definable procedure prb_print_menu, which is given the message list and the options list as its arguments.

The user responds by typing in one of the menu options, or one of the interactive commands described below (e.g. ".why", ".data" etc.). If no

acceptable response is provided the program prints the menu out again.

-- User-defined action keywords and prb_action_type ------------------

Besides the action types described above (e.g. [NOT ...] [DEL ...] [POP11 ...] [MENU ...] users can define new complex action types associated with new keywords, by creating a procedure to be run in the action, and using the property prb_action_type to associated the keyword with that action.

-- . Example of a new action keyword

To define a new action keyword "REPORT" do something like this:

```
 define doREPORT(rule_instance, action);
 ....
 enddefine;

 "doREPORT" -> prb_action_type("REPORT");
```

Then any action starting with the word REPORT will invoke this procedure, by applying it to the current rule_instance (i.e. the current activation of the rule containing the action that starts with the keyword) and the instantiated action from the rule's action list.

Storing the name, rather than the procedure, in the property prb_action_type allows the procedure to be redefined, or traced, during program development, without the need to replace the old version in the property.

The test for whether an action is user-defined is done before recognising built-in types. This makes it possible for a user-defined type to over-ride the built in action type. I.e. after instantiating the action, as described above, the program first looks to see whether the initial word of the action has been associated with a procedure using the user-assignable property prb_action_type, in which case the procedure is run with the current rule and the current action as arguments. Otherwise, it checks to see whether the action is one of the built-in forms listed below.

 -- NEWLIMIT action --

This type of action was requested by Peter Waudby. It is not built in but is an autoloadable extra, available for use with rulefamilies.

The command

```
 uses new_family_limit
```

makes available the action form

```
 [NEWLIMIT <integer>]
```

This allows the current cycle limit associated with a rulefamily to be changed dynamically. It may later be supplanted by more general mechanisms for run time manipulation of rulefamilies. This will probably require a change in the implementation of cycle limit mechanisms.

-- Interactive commands available during READ, MENU or Tracing pauses

During execution of READ actions and at interaction pauses resulting from -prb_walk- being true or a rule being traced (as described below), the user has the option of typing in any of the following interactive commands, all of which start with a dot or colon to indicate that what is typed in is not an answer to a question. (The full effects of these commands cannot be understood without reading later sections of this file describing the effects of variables like prb_show_conditions, prb_chatty, etc.

-- -- .show
 This causes information about the current rule and the current action from that rule to be printed out. The printing is done by
 the user-definable procedure

 prb_displayrule(action, rule_instance)

 which is given the current (instantiated) action, and the current rule instance from which it comes. (The form of a rule instance is defined above.)

-- -- .why
 This causes an explanation of the current action to be printed out, if provided (i.e. in a READ action, as explained above). If "why"
 is typed again, repeatedly, then information about previously activated rules is printed out, until there are no more rules.
 This "repeated why" option works only if the user has made the global variable prb_remember a list initially (e.g. []), rather than false. Otherwise the information about previous interactions will not be stored for interrogation in this way.

 For the current READ action "why" will cause a user-provided explanation string to be printed out, if it is given in the fourth element of the READ action. Otherwise prb_displayrule is used
 to print out the action and the context, in answer to "why".

-- -- .data
 .data
 This causes information in the working memory, prb_database, to be printed out.

 .data <pattern>
 This causes all the data that match the pattern to be printed out.

-- -- .trace and .untrace
 .trace <rule names>
 Causes tracing to be switched on for the named rules, like prb_trace, described below.

 .untrace <rule names>
 Causes tracing to be switched off for the named rules, like prb_untrace, described below.

 .untrace all
 Causes tracing to be switched off for all rules, and prb_walk
 made false.

-- -- .show_conditions
 .show_conditions true
 .show_conditions true Causes prb_show_conditions to be made true, so that testing of
 all rule conditions is shown in detail.

 .show_conditions false
 Turns the above off

-- -- .chatty

 .chatty
 .chatty true
 Either of these makes prb_chatty true

 .chatty false
 This makes prb_chatty false

 .chatty <integer>
 This assigns the integer to prb_chatty

-- -- .walk

 .walk
 This makes prb_walk true

 .walk false
 This makes prb_walk false

-- -- :<Pop-11 expression>
 If the user types a colon followed by a Pop-11 expression, then the Pop-11 expression is executed and if there is any result it is printed out. This may be useful during debugging.

-- -- ? (or any unrecognised command)
 This causes a help message to be printed out, listing the commands available.

-- Actions are instantiated before they are run ----------------------

Before rule actions are executed they are instantiated. This means that any variables preceded by "?" or "??" that have been bound in the conditions of the rule are replaced by their values. Unbound variables and their prefixes are left uninstantiated. In addition "popval" and "apply" list elements (described below) are evaluated and their results, if any, used to replace the list.

-- . Variable bindings in actions

Before any action is executed it will have been instantiated in a rule-instance. This is because there may be different ways of satisfying the conditions in a rule, and these will correspond to different action instances.

Expressions defining actions may include variables (preceded by "?" or

"??", but without the use of restrictions permitted in patterns -- e.g. "?x:isword" has no role in an action, though it can occur in a condition).

When a rule is instantiated the variables will normally have been bound by testing the conditions of the rule against the working memory (prb_database). Normally, no variable should be used in an action unless it also occurs in a simple condition, since it is only from a satisfied simple condition that a variable can acquire a value.

However in more complex rules, some of the variables may be set by [VARS....] or [LVARS...] forms or [POP11...] forms.

In the actions in a rule-instance, any such variables will be replaced by their values before the actions are performed. (See the section on instantiation of actions, below. Whether a variable preceded by "?" or "??" is instantiated depends on whether the variable is in the list popmatchvars.)

Complex conditions like
 [NOT ...]
 [WHERE ....]
 [NOT_EXISTS ....]

cannot bind variables. The exception is an OR condition or ALL condition, which can.

Actions may also include "popval" elements or "apply" elements, to be evaluated before the action is performed, as described below.

-- . "popval" or $$ list elements

Any element of an action, embedded at any depth may be a list of the form:

 [popval <Pop-11 expression>]
 [$$ <Pop-11 expression>]

i.e. a list whose first element is "popval" or "$$".
 In early versions of Poprulebase these expressions were compiled at run time. Besides being inefficient, this is not compatible with the use of lexical variables, so since Poprulebase version 4.0 the expressions are compiled when the rules are read in. In other words they are replaced by procedures. Then what happens at run time is that the procedure is run and any resulting value is used to replace the complete list. If there is no value the list is ignored, though side effects may have occurred.

This means that these forms must not use "?" or "??" before pattern variables. E.g. this condition could bind the variable x:

 [age ?person ?x]

then, in a later condition or action the following might be embedded:

 [....[$$ x + 5] ...]

Then if x gets the value 3 from the condition, the embedded list [$$....] will be replaced by 5.

E.g. the following could be an action.

```
 [remember [$$ x + y]]
```

x and y should already have values. If the values were 2 and 3 respectively then this action would add the following to the working memory.

```
 [remember 5]
```

Notice that the value of the expression is not put in an embedded list, unless the expression evaluates to a list. E.g. an action of the form

```
 [... [$$ 10//3 ] ...]
```

evaluates to

```
 [... 1 3 ...]
```

not to

```
 [... [1 3] ...]
```

-- "apply" or $: list elements --

The popval mechanism is very general because it allows an arbitrary Pop-11 expression to be compiled and run.

However, because it requires the form of an expression to be known at compile time. In other cases it may be preferable to use the [apply...] form, which can be abbreviated to [$: ....]. This requires the word "apply" or "$:" to be followed by a procedure or name of a procedure, and then arguments for the procedure.

An "apply" list element is of the form

```
 [apply <procedure name> <arg1> <arg2> .... ]
```

or, equivalently

```
 [$: <procedure name> <arg1> <arg2> .... ]
```

Variables that are to be instantiated MUST be preceded by either "?" or "??". In the latter case the value the variable must be a list and the whole list will be spliced in before evaluation of the whole expression. This is what makes these forms a little more flexible than [popval....] in some contexts.

The named procedure will be applied to all the arguments. So in the previous example, instead of the action

```
 [remember [popval x + y]]
```

use

```
 [remember [apply + ?x ?y]]
```

or

```
 [remember [$: + ?x ?y]]
```

And instead of

```
 [... [popval 10//3 ] ...]
```

use

```
 [... [apply // 10 3]...]
```

or

```
 [... [$: // 10 3]...]
```

Summary:
In an action, a popval item (or $$ item) is treated as arbitrary Pop-11 code, which is run, whereas an apply item (or $: item) is treated as a procedure application expression which is interpreted by putting the arguments on the stack and calling the procedure.

```
-- Rule manipulating procedures ----------------------------------------
 -- Global variables ---------------------------------------------------
```

There are several global variables used by the program. Users may sometimes wish to access them. Procedures for manipulating them are described later.

```
-- -- prb_database <the current database>
```

This is used as the working memory when rules are running. It is a property created by newproperty, or the procedure prb_newdatabase, which calls newproperty. The property associates keys (usually words that occur as the first item of a database entry) with lists of database items that all start with the same key. Applying the property to a key gives access to the list.

Lists of items starting with a given key can be accessed through code of the form

```
 database( key ) ==>
```

e.g.

```
 prb_database("status")==>
 ** [[status b1 passive] [status b2 active] [status b3 dead]]
```

The second argument of prb_run determines the value of this global variable, and it is accessed by a collection of procedures partly analogous to the Pop-11 database procedures described in

```
 HELP * DATABASE.
```

```
-- -- prb_rules <a list of rules>
```

This is a list of rules usually created by using the form

```
 define :ruleset .... enddefine
```

as defined above. Additional individual rules can be appended using the syntax

```
 define :rule .... enddefine
```

as described above. It is possible in principle to manipulate several lists of rules, by temporarily saving and restoring the value of prb_rules, or by using a rulefamily. See HELP * RULESYSTEMS

The procedure

```
 prb_rule_named(<word>, <list>) -> <rule>
 prb_rule_named(<word>) -> <rule>
```

Searches down the given <list> for the rule with the name. If not provided, <list> defaults to prb_rules.

A list of the form of prb_rules is required as first argument to prb_run.

When prb_run is executing the current ruleset is held in the global variable prb_rules, which is dynamically local to the procedure prb_run_with_matchvars.

-- -- prb_ruleset_name <the name of the current ruleset>
 If pop_debugging is true the interpreter will attempt to ensure that the word which is the name of the current ruleset is held in this variable. Whether this is possible or not will depend on how rulefamilies and rulesystems are constructed. If a ruleset list is given to prb_run directly or if a rulesystem or rulefamily points to the list directly instead of via the name, then the interpreter may not have the information to assign a word to prb_ruleset_name.

-- -- prb_family_name: false or word

If the current rulecluster has a name, this variable will hold that name. It is dynamically local to prb_run_with_matchvars

-- -- prb_current_family

If a rulefamily is currently active (see HELP * RULESYSTEMS) then this variable holds that rulefamily (a prb_rulefamily record) as its value.

Otherwise its value will be false.

-- -- prb_current_rule_prop

If a rulefamily is currently active (see HELP * RULESYSTEMS) then this variable olds a property which provides the name to ruleset mapping in the current rulefamily.
 prb_family_prop(prb_current_family) -> prb_current_rule_prop

It should probably not be accessed by users.

-- -- prb_max_keys <a number>

This variable controls the default size of the property table used for the database. The default is 64. If the program uses a much larger number of database keys then there could be efficiency gains by increasing this

number. See prb_newdatabase

-- -- prb_noprint_keys <list of words>

This is explained in connection with prb_print_table, below. It prevents certain database items being printed out by the standard printing mechanisms.

-- -- prb_actions_run <counter variable, or false>

This can be set to 0 before the rule interpreter is run. It is then incremented whenever an action other than STOP, STOPIF, QUIT, QUITIF, STOPAGENT or STOPAGENTIF action.

Note;
 STOPAGENT and STOPAGENTIF actions are defined in LIB SIM_AGENT.
 See HELP * SIM_AGENT

At the end of a run of the interpreter, if prb_actions_run is unchanged, that means no non-trivial actions were run. For more on the use of this see HELP * SIM_AGENT.

-- -- prb_trace_ruleschanged <boolean>

The default value is false. If this variable is made true, then if there is a change of ruleset during a run of prb_run, then the user definable trace procedure
 prb_ruleschanged_trace(newruleset, rulefamily);
is invoked. By default the procedure does nothing. See occurrences of prb_trace_ruleschanged in
 LIB * poprulebase

-- -- this_rule <a rule>

This variable is given the current rule being checked by prb_applicable or the rule whose action is being performed by prb_do_action_now
which implements the [DO <action>] format.

Useful for tracing and debugging.

-- -- this_rule_name <a word>

This variable is given the name of the current rule being checked by prb_applicable. Useful for tracing and debugging.

-- -- do_trace_match <boolean>

If made true before poprulebase is compiled, this enables additional tracing and profiling to be done. It can use used after that to turn the tracing on and off. See prb_trace_procs, below.

-- -- word_of_ident {property}
This is used to get from an identifier to the word which has the identifier. It is used in the procedure print_ident, used in sys_print_ident

-- Procedures relating to the database --------------------------------

Because prb_database is not a list, its elements cannot be accessed directly. So the following procedures are provided to manipulate databases of this type.

-- . prb_newdatabase(hashlen, items) -> newdb
 Given an integer, for a property table size and a default list of database items return a new database containing the items.

 See prb_max_keys

-- . prb_print_table(database)
-- . prb_print_table(database, keys)
 Prints the entire contents of the property table, using ==> to print each of the lists of items sharing a common keyword. The second form, which uses an optional list of keys, constraints the procedure to print out only the lists of items starting with those keys, and to print them in the order given in keys.

 Both procedures ignore keys which are in the list prb_noprint_keys. That is because the SIM_AGENT library adds extra items to the database which are part of its implementation of rulesystems. Those items should not normally be printed out when what is required is information about changing database contents.
 See HELP NEWKIT (part of Sim_agent).

-- . prb_print_database( )
 This prints out the database currently held in prb_database. It uses prb_print_table(prb_database);

-- . prb_add(item)
 Adds item to prb_database
Related procedures follow

-- . prb_add_db_to_db(db1, db2, copying);
Add all elements of db1 to db2. If copying is true, copies are added. Otherwise the originals are used, and non-constructive concatenation causes the contents of db1 to be corrupted.

-- . prb_add_to_db(item, dbtable);
Add one item to the property table.
 -- . prb_present(pattern) -> false or item
 Returns false or the first item in prb_database matching pattern
 If an item is found it is also assigned to prb_found. (NB use prb_in_database for the equivalent of the Pop-11 present
 procedure).

 Warning:
 This procedure is not like the Pop-11 database procedure present, because it does not localise popmatchvars. Thus if it is invoked with a pattern containing a variable that is already in the list popmatchvars, then that variable behaves like a constant, i.e. its current value is used, and it will not be reset during matching. This is to enable the behaviour of prb_present in a complex
 rule to be governed by the variables bound in the rule's conditions. See HELP * SYSMATCH

-- . prb_present_keys(pattern, keys) -> item;

 Like prb_present, but specify database keys to use

-- . prb_in_database(pattern) -> false or item
 This is exactly like prb_present, except that it includes dlocal popmatchvars = [];
 This is the procedure that should be used in place of the Pop-11 database procedure, present. See HELP * PRB_DATABASE

-- . prb_del1(pattern, data) -> (item, data);

 Delete the first item in the list data that matches pattern.
 Return the item found, or false, and the new (or unchanged) list Assume data is a list, so use fast_for. Used by prb_flush1

-- . prb_flush(pattern)
 Removes everything in prb_database that matches the pattern. If prb_copy_modify is false then the removal is non-constructive,
 i.e. database list links are re-used (to save garbage collections).

 if DATABASE_CHANGE (see below) divides prb_chatty, then on exit the variable prb_found holds a list of the items removed.

-- . prb_flush1(pattern)
 Like prb_flush except that it stops after removing one matching
 item from the database (and puts it into a list assigned to prb_found)

-- . prb_match_apply(dbtable, pattern, proc) Apply the procedure proc to every item in dbtable matching the pattern. Note that this can set popmatchvars as it uses sysmatch.
 It will also use popmatchvars to control sysmatch, so the value of popmatchvars has to be reset before each call of sysmatch. Thus
 most procedures using prb_match_apply should do

 dlocal popmatchvars = [];

For example, it can use used to define prb_finish (whose role is explained elsewhere) so as to make prb_run leave on the stack a list of all the data items patching a given pattern, thus:

 define prb_finish(rules, database);
 ;;; Note the rules will be ignored
 lvars rules, database;
 dlocal popmatchvars = []; ;;; reset popmatchvars on exit

 [%prb_match_apply (database, [.....], identfn) %]
 enddefine;

Replace [....] with the desired pattern. or #_< [....] >_# if the pattern is not to be re-created each time.
 (See HELP * HASH_ and REF #_<)
 Note, since popmatchvars is a temporary list, it can be returned to free store at the end of prb_finish, as described in REF * FASTPROCS e.g.

 sys_grbg_list(popmatchvars)

-- . prb_match_apply_keys(dbtable, pattern, keys, proc)
Apply the procedure proc to every item in dbtable starting with one of the keys, which matches the pattern.
See note about popmatchvars in prb_match_apply

-- . prb_remove_all(list_of_patterns) -> found

Removes all the items from prb_database that matches any of the patterns in the list, and returns a list of them.

-- . prb_forevery(patternlist, proc)

This is the main procedure used for checking whether the conditions of a rule are satisfied, and for creating all instances of a rule if some conditions can be satisfied in more than one way.

proc(vector, integer)
is a procedure that takes two arguments, a vector and a number, used for recording "relative recency" information about the items satisfying the conditions of an action, within the prb_recof field of a rule instance.
The vector is a vector of integers giving the relative locations of database items satisfying conditions in patternlist. The positions are relative ONLY to other items in the same sub-database of prb_database.
E.g. if the vector is { 3 1 9 0 ....} and the number is 4, then that says that the first condition matched the third most recent item in its sub-database, the second condition matched the most recent item, in its sub-database the third one matched the 9th most recent and the fourth one was a "complex" condition not matched by a single database item, e.g. a WHERE, NOT or ALL condition.
(See the information about prb_recof, above.)

For every possible way of matching the patterns in patternlist prb_forevery runs the procedure proc with the pattern variables set. This is used to find all possible ways of instantiating the conditions of each rule, in order to build a list of possible applicable rules, to be sorted by prb_sortrules.

However, this process will be cut short if

there is a [CUT ...] condition at the end of patternlist
or
prb_allrules (described below) is false

-- . prb_allpresent(patternlist) -> false or found_list
This is defined as follows

```
define prb_allpresent(patternlist) -> found;
lvars patternlist, found;

define lconstant procedure report_success( vec, num );
;;; return the items found.
lvars vec, num;
ncrev(prb_found);
exitfrom(prb_allpresent)
enddefine;
```

```
 prb_forevery(patternlist, report_success);
 false -> found;
 enddefine;

-- . prb_empty(dbtable) -> boolean
 Return true if there's nothing in the table, otherwise false

-- . prb_storedata(<filename>)

prb_storedata(<filename>);
 Stores the current poprulebase database in a format which can later be
read and edited, or recompiled to create a new database.

-- Additional procedures and variables used for tracing

-- . prb_show_conditions (boolean or list of words)

If -prb_show_conditions- is true then whenever a rule is about to be
tested, its name and all its conditions are printed out, then as each
condition in turn is checked the result is printed. If it is a list, then
this happens only for rules whose names are in the list. If it is false,
then the tracing of condition testing is turned off.

The printout can be a little confusing because poprulebase will try all
possible ways of matching conditions to the database. Indentation
indicated by vertical bars is used to show the dependencies.

If the conditions are C1, C2, C3, then information about whether C1 is
satisfied is prefixed with "|". After C1 has been satisfied, information
about C2 is printed out preceded by "||". If C2 is satisfied, then each
information about satisfaction or non-satisfaction of C3 is prefixed with
"|||".

After that, any further reports on tests for a different way of satisfying
C2 will be prefixed with "||". When all those have been exhausted,
additional ways of satisfying C1 will be tested, and results prefixed with
"|". If any are successful then new attempts will be
made on C2, prefixed with "||", and so on. Thus the vertical bars show the
`backtracking' during testing of conditions. (Note that the [CUT])
condition form can be used to prevent such backtracking.)

For every combination of conditions that is satisfied, the variables in
the conditions will be printed out with the values assigned to them by the
matcher in satisfying the conditions. (See example below.)

Further trace information is controlled by prb_chatty and prb_walk.


-- . . Example of prb_show_conditions trace output

The file TEACH PRBRIVER describes a poprulebase program that makes a plan
for getting man, fox, chicken and grain across a river. One of the rules
is called "complete_move" in the rule_set called "solve_rules". It has two
conditions to be satisfied: [complete_move ?move] and [state ?state]

If prb_show_conditions is set true the following illustrates the printout
when this rule is tested for applicability:
```

```
------------------------------------------------------------------------

** [Checking conditions for: complete_move in solve_rules] ;;;rule
** [[complete_move ? move] [state ? state]] ;;;conditions |** [SUCCESS
[complete_move [move chicken]]]
||** [SUCCESS [state [[chicken isat right]
 [fox isat left]
 [grain isat right]
 [man isat right]]]]
|||** CONDITIONS SATISFIED: Variables bound:
|||** state = [[chicken isat right] [fox isat left] [grain isat right]
[man isat right]] ; move = [move chicken] ;

------------------------------------------------------------------------
 In this case no other ways of satisfying the conditions will be attempted
because prb_allrules is set -false- and the first applicable rule instance
is therefore selected.

A more complex example showing the attempt to find different ways
of satisfying the conditions follows. The rule being tested is
"move_thing" in the rule_set "solve_rules". It has seven conditions,
one of them an OR condition, whose first disjunct fails, while its second
succeeds in one case. The program tries to find something
at the same side of the river as the man (in this case the right
side). The first candidate is the man, but this causes the
thing /= "man" test to fail. The second candidate is the grain, and
that fails because the last move in the history list was moving
the grain, so it eventually tries the third candidate, the chicken,
and the rule is shown to be applicable, with "thing" bound to "chicken",
etc.

------------------------------------------------------------------------ **
[Checking conditions for: move_thing in solve_rules]
** [[man isat ? place]
 [? thing isat ? place]
 [WHERE thing /== " man "]
 [OR [opposite ? place ? other] [opposite ? other ? place]]
 [state ? state]
 [NOT tried [move ? thing] ? state]
 [NOT history [[move ? thing] =] ==]]
|** [SUCCESS [man isat right]]
||** [SUCCESS [man isat right]] ;;; trying thing = "man" |||** [Tested
WHERE [thing /== " man "]]
|||** [Result is: <false>]
||** [SUCCESS [grain isat right]]
|||** [Tested WHERE [thing /== " man "]]
|||** [Result is: <true>]
||||** [SUCCESS [opposite right left]]
|||||** [SUCCESS [state [[chicken isat right]
 [fox isat left]
 [grain isat right]
 [man isat right]]]]
||||||** [SUCCESS [NOT tried
 [move grain]
 [[chicken isat right]
 [fox isat left]
 [grain isat right]
```

```
 [man isat right]]]] |||||||** [FAILED [NOT history [[move grain] =] ==]]
|||||** [FAILED [state ? state]]
||||** [FAILED [opposite ? place ? other]]
||||** [FAILED [opposite ? other ? place]]
||||** [FAILED [OR [opposite ? place ? other] [opposite ? other ? place]]]
||** [SUCCESS [chicken isat right]] ;;; trying thing = "chicken" |||**
[Tested WHERE [thing /== " man "]]
|||** [Result is: <true>]
||||** [SUCCESS [opposite right left]]
|||||** [SUCCESS [state [[chicken isat right]
 [fox isat left]
 [grain isat right]
 [man isat right]]]]
||||||** [SUCCESS [NOT tried
 [move chicken]
 [[chicken isat right]
 [fox isat left]
 [grain isat right]
 [man isat right]]]]
|||||||** [SUCCESS [NOT history [[move chicken] =] ==]]
||||||||** CONDITIONS SATISFIED: Variables bound:
||||||||** state = [[chicken isat right] [fox isat left] [grain isat
right]  [man isat right]] ; other = left ; thing = chicken ; place = right
 ;
```

As this example illustrates, making -prb_show_conditions- true can produce an enormous amount of printout. But this is sometimes essential for debugging.

-- . prb_show_ruleset {boolean}

If this is true, show name of ruleset when starting a run of the rule interpreter or when switching to a new ruleset.

-- . prb_chatty {boolean or integer}

The variable prb_chatty controls trace printing while rules are running, as follows, depending on whether its value is false, true, or an integer.

This is copied from LIB * POPRULEBASE :

```
lconstant
 INSTANCES = 2,
 WHERETESTS = 3,
 DATABASE = 5,
 APPLICABILITY = 7,
 APPLICABLE = 11,
 DATABASE_CHANGE = 13,
 SHOWRULES = 17,
 TRACE_WEIGHTS = 19,
;
```

How the value of prb_chatty affects behaviour of poprulebase

 if false, there is no trace printing of the sort described here (though rules can print things out using SAY actions, and specialised trace

procedures can do their own printing.)
 if true, causes minimal useful information to be printed out during running
 if the value is divisible by 2 (INSTANCES) then the list of rule-instances already activated is printed out on every cycle, if prb_repeating is false and the list is to be remembered.
 if the value is divisible by 3 (WHERETESTS) then all WHERE and POP11 tests are printed out
 if the value is divisible by 5 (DATABASE) then the database is printed out on every cycle
 if the value is divisible by 7 (APPLICABILITY) then checks on applicability conditions for rules are printed out.
 (Compare prb_show_conditions)
 if the value is divisible by 11 (APPLICABLE) then the list of applicable rules found is printed on every cycle
 if the value is divisible by 13 (DATABASE_CHANGE) then information is printed out concerning items added to or removed from the prb_database.
if the value is divisible by 17 (SHOWRULES) and prb_walk is false then every rule that is activated is printed out, without an interactive pause.
 if the value is divisible by 19 (TRACE_WEIGHTS) then information about weights of rules is printed out, if weights are used (i.e. if prb_useweights is true).

 The default value of prb_chatty is false.

For example, to make checks on applicability conditions printed out, and every rule that is activated printed out do

```
 7 * 17 -> prb_chatty;
```

The value of prb_chatty can be altered interactively as explained above. the procedure
 prb_divides_chatty(int) -> boolean;
may be useful for checking whether a particular number is already included in prb_chatty.
It is defined in LIB POPRULEBASE/prb_divides_chatty

-- . prb_repeating {boolean}

Default is true.

If this is set false the system will not trigger the same rule on the same prb_database items twice. Use of this option requires all rules that are fired to be remembered in association with the database items that triggered them. The remembered rule-instances are stored in the list prb_remember.

The actual database items are remembered for comparison with future items that make the condition in the rule true. This means that if rule R1 runs once with database item [r a b] making its condition true, then it may run again later if a new item [r a b] is added to the database.

Setting prb_repeating false adds to the memory load of the program as well as requiring additional tests. See also the WARNING in connection with

prb_copy_modify.

In some cases, careful ordering of rules can make it unnecessary for prb_repeating to be made false.

NOTE: if you are using poprulebase with the SIM_AGENT library, then making prb_repeating false may, cause a large garbage overhead, and not always work as expected. So it will generally be better to use special database entries and conditions in rules to prevent a rule being run again on data that have already been processed.

-- . prb_walk {boolean}
-- . prb_walk_fast {boolean}

If prb_walk is set true then before each action is performed, the action and the rule it is in will be printed out, using prb_interact.

If prb_walk_fast is false (the default) the procedure prb_interact will pause and allow questions to be asked. In particular

 ".show"
 displays the current rule and the action from that rule.

See additional information about Interactive commands above.

In order to distinguish pauses due to prb_walk being true from the pauses when real questions are being asked, e.g. using the READ action format, a different prompt is used, namely:

 Walking>

The default value for prb_walk is false.
The default value for prb_walk_fast is false.

To allow actions to be traced with pausing make only prb_walk true.
To allow actions to be traced without pausing make them both true.

-- . prb_pausing {boolean}

If this is not false then PAUSE actions will work. If it is false they are ignored. (Note this does not affect prb_walk.)

-- . prb_explain_trace {boolean}

If this is true then all [EXPLAIN ...] actions will be carried out

Default is true.

-- . prb_debugging {boolean}

If true, extra information is printed out, and Pop-11 error messages that are caused in WHERE conditions are not suppressed. Default is TRUE.

-- User-definable procedures for self-monitoring ----------------------

In August 2000 a collection of additional tracing procedures was added to Poprulebase, partly suggested by Catriona Kennedy. These supplement the above printing or interactive tracing facilities by enabling the machine to record within itself some of the steps in its operation.
E.g. an agent could remember which conditions it has recently checked, or which items it has recently added to the database.

By default these trace procedures are all turned off. They can be made true by making the global variable prb_self_trace true. (Using DLOCAL this can be done on a rule by rule, or ruleset by ruleset
basis or for individual agents).

In each case the first argument agent, will be the current value of sim_myself, which defaults to undef, but can be the current objectlass instance if the sim_agent toolkit is used. That means that these procedures can be redefined as methods that do different things for different classes of objects or agents.

```
-- . prb_checking_conditions_trace(agent, ruleset, rule);
 ;;; invoked when about to check conditions, before call of prb_forevery

-- . prb_checking_one_condition_trace(agent, condition, rule);
 ;;; invoked in prb_forevery_sub just before the condition is processed

-- . prb_all_conditions_satisfied_trace(agent, ruleset, rule,
matchedvars);  ;;; invoked if all conditions satisfied.
 ;;; matchedvars is the list of variables bound

-- . prb_condition_satisfied_trace(agent, condition, item, rule,
matchedvars);  ;;; invoked when one condition is satisfied.  ;;;;
matchedvars is the list of variables bound at the time
 ;;; if the condition is a simple pattern, then item will be the item in
the  ;;; database that matched it. Otherwise item may be undef, e.g. for
;;; an [OR ...] or [NOT ...] or other complex condition.

-- . prb_doing_actions_trace(agent, ruleset, rule_instance);
 ;;; invoked when about to run actions of rule after conditions checked

-- . prb_do_action_trace(agent, action, rule_instance;
 ;;; invoked in prb_do_action with each individual action
 ;;; just before it is performed.

-- . prb_adding_trace(agent, item);
 ;;; invoked when adding item to the current database

-- . prb_deleting_trace(agent, item);
 ;;; deleting one item from the current database

-- . prb_deleting_pattern_trace(agent, deleted, pattern);
 ;;; Deleting everything that matches the pattern, e.g. prb_flush
 ;;; deleted is a list of the items deleted. May not be available if  ;;;
the global variable prb_recording is false (the default)
 -- . prb_modify_trace(agent, item, action, rule_instance);
 ;;; action is a list of form [MODIFY <item> <key> <value> <key> <value>
...]  ;;; where <item> should refer to item, though it may be a number
 ;;; it may be necessary to apply prb_instance to the action to find  ;;;
the actual values.
```

-- . prb_condition_failed_trace(agent, condition, rule);
 ;;; invoked in many places where a condition has not succeeded.
 ;;; It is difficult to ensure that all of them are captured.

-- . [prb_pattern_matched_trace(agent, pattern, item);]
 ;;; not yet used, in case prb_condition_satisfied_trace
 ;;; suffices. There are many additional places where the
 ;;; matcher is invoked, and it is not clear whether they should
 ;;; all be monitored.

-- . Controlling trace procedures: prb_tracing_on, prb_self_trace

Each invocation of any of the above procedures is preceded by "IFTRACING".
This means that if the pop-11 variable prb_tracing_on (default true) is
made false before LIB poprulebase is compiled, the trace procedure calls
will not be included in the code.

If that variable is true at compile time then the global variable
prb_self_trace (default false) can be made true to turn on the trace
procedures. This can also be done in [DLOCAL ...] expressions in
individual rules or rulesets, as mentioned above.

-- . Use of sections IFSECTIONS {boolean}

The default value is false.

If the global constant IFSECTIONS is made true BEFORE poprulebase is
compiled then various bits of code will be enabled which allow rulesets to
be compiled in different sections.

-- . prb_rulesection {property}
This property is used to associate a rule with the section in which it was
compiled.

-- . prb_use_sections {boolean}
Default is false.

If IFSECTIONS is true at compile time this variable can be used to turn
use of sections on or off.

(The stuff on sections is a mess, and as far as I know has never been
required. It may be useful in a big project. But lexical scoping has
reduced the requirement.)

-- . prb_check_section(rule);
System utility used in connection with sections.

-- Additional control variables --

-- . prb_allrules {boolean, or 1}

If true, then on every cycle of the rule interpreter, all the rules whose
conditions are true will have their actions executed, not just the first
rule found. If the value of prb_allrules is false, or 1, then
only actions from the first rule found to be satisfied will be run.
Moreover if its conditions can be satisfied in more than one way, only one

instance of the rule will be created and run. If prb_allrules is true, then on each cycle all possible instances of all rules with satisfied conditions will be run.

E.g. if the list of conditions of a rule can be satisfied in more than one way (e.g. the pair

```
 [father ?a ?b][mother ?b ?c]
```

may have several instances), then if prb_allrules is true, then all the instantiations of the rule will be found by prb_applicable on each cycle.

How they are run depends on the value of prb_sortrules. By default this has the value false. However, if its value is a procedure, it should be a procedure that takes a list of rule instances as input and returns a list of rule instances (possibly empty). prb_sortrules can be defined to reorder the rule instances, or filter some out, as explained below.

If prb_sortrules is false (the default value) then the list of rule instances is not created, and the actions of the corresponding rules are run as soon as the conditions have been found to be satisfied.

Trace -prb_applicable- to see which applicable rule instances are found on each cycle. This will not work if prb_sortrules is left false. The simplest way to change it so that a list of runnable rule instances is created is to do

```
 identfn -> prb_sortrules;
```

Even then if prb_allrules is false, then only the first possible rule instances is run on each cycle of the interpreter, without creating a list of options.

If the value of prb_allrules is the integer 1, only one rule will be run on each cycle. Nevertheless, all applicable rules will still be given to prb_sortrules, and the first instance, after sorting, will be run.

This can be useful for debugging - in order to show all the applicable rules, even if only the first one is to be run.

Alternatively prb_allrules can be made true, and prb_sortrules can be made to return a list of only one rule.

The default for prb_allrules is false. This means that only the first applicable rule that is found will be executed on each cycle. If the rule has more than one applicable instance (i.e. more than one way of matching the conditions against items in working memory), it is not defined which one will be selected.

-- . prb_recency {boolean}

Since the changes to poprulebase in July 1995 this variable is only of limited value, and may be withdrawn.
 If it is true, then after the checking of the conditions of a rule, additional information is stored in the prb_recof field of each rule instance of that rule, representing the relative recency of addition to

the database of all database items matching the conditions.
This information can be used by prb_sortrules to select rules according to how recently they have been made true. However, the recency information for each item is relative only to other items with the same database key, because the absolute order of addition of items to the database is not recorded.

Thus if the conditions C1, C2 and C3 match the fourth the first and the seventh items of the relevant sublists in prb_database, then the recency representation will be a vector of three numbers

 {4 1 7}

However, C1 may be newer than C2 if the database items starting with the same key as C1 have been changing more recently.

If a rule condition is a complex condition, e.g. of type NOT or WHERE, LVARS, VARS, or POP11, no particular database item makes it true, so the corresponding number associated with that condition in the rule instance will be 0.

-- . prb_useweights {boolean}

If this variable is true then after the list of possible rule activations has been constructed, and after the (user-definable) procedure prb_sortrules has been applied to the list, a selection by weight is made. I.e. the instance of the rule with the highest weight is selected and all others ignored. Thus, if prb_useweights is set true it is assumed that only one rule will be run on each cycle.

If there are several rule-instances with the same maximal rule weight, then the first one is selected.

Users wishing to employ weights in some more sophisticated matter can define the procedure prb_sortrules accordingly. E.g. It can return all rule instances whose weights are above some threshold, all of which are to be run. In that case prb_useweights should be set false.
 Note that prb_print_rule will show the weight of a rule only if prb_useweights is true.

-- . prb_sortrules {false or procedure}

The default for this is false. If non-false, it is assumed to be a user-defined procedure that sorts rules that are applicable. This makes sense only if prb_allrules is non-false.

prb_sortrules is given a list of possible rule activations and returns a list of possible rule activations.

A possible rule activation is represented as a vector consisting of four or five elements. If prb_recency is false there are only four elements in each rule, namely:

 - a rule whose conditions are satisfied
 - a list of variables bound when the conditions were satisfied,
 - a list of the corresponding values of the variables.
 - a list of the database items matching the conditions

If prb_recency is true then there is an extra element, a vector of "recency information" as described above in connection with prb_recency. Example definitions of prb_sortrules, corresponding to different search strategies are given below.

-- . prb_remember {false or list}

If this is non-false, then in prb_run it is given a list as value, and then all activated rule-instances are added to the front of the list, in the form of a rule_instance record of the type described under prb_sortrules.

The Default is false.

prb_remember is used in connection with requests for explanations. (A very primitive explanation facility). It is also used if prb_repeating is false.

-- . prb_copy_modify {boolean} (DANGER)

This defaults to true. If it is set false, then changes to the database are made without copying. I.e. the list-links in the database are re-used, reducing the frequency of garbage collections.
(See HELP * EFFICIENCY)

This applies in particular to prb_flush and to MODIFY, REPLACE and DEL actions.

WARNINGS:
(a) if this variable is set false then it is possible for data saved by one rule to be corrupted by the action of another. (b) this variable cannot be set false if prb_repeating is set false, since the latter requires database items to be remembered.

The program attempts to minimise potential damage caused by making this variable false. In particular, if a rule adds something constant to the database then if prb_copy_modify is false, then a copy of the constant list is added, since otherwise later modification of the list could alter the rule itself!

-- . prb_prwarning {procedure}

The value of this variable is assigned to the Pop-11 variable prwarning while the rule interpreter is running. It determines how warnings about automatically declared variables should be printed. See HELP * PRWARNING

prb_prwarning defaults to erase, so that no warnings are given, but could be given sysprwarning as its value, in which case undeclared identifiers will be announced when they are declared.

-- . prb_get_input {boolean for asynchronous input}

If prb_get_input is true, then if user types anything during execution the whole line (terminated by RETURN) is read in as a list using readline() and added to the working memory. Then if appropriate it may trigger a rule on the next cycle. The test for whether there is any input waiting to be

read in is carried out after each rule is activated.

For example, suppose the variable is true, and the user types a line consisting of the character "d", followed by a space, followed by additional words etc, then if there is a rule of the form below it will be activated before the next rule, if selected by prb_sortrules:

```
 RULE process_input
 [d ??rest]
 ==>
 ;;; delete the item
 [DEL 1]
 [SAY message ??rest received]
 [READ 'OK?' []]
```

This will pause on the READ action, allowing the user to interrogate the database, etc.

-- . prb_memlim {false or integer} WITHDRAWN

No longer available, since the conversion to a property table for the database.

-- . prb_max_conditions {integer}
 This integer defaults to 30 and represents the maximum number of conditions that a rule can have. It is used only if prb_recency is true. If it is changed lib poprulebase will have to be re-compiled. However, it can be set before lib poprulebase is compiled.

-- . prb_add_condition_vars(list);
-- . prb_add_action_vars(rule_instance, action);
(Nothing to do with prb_add).

These are used by "VARS" and "LVARS" actions and conditions to ensure the variables are in popmatchvars. (For system use only.)

-- Some user definable procedures ------------------------------------

-- . prb_eval(action)
-- . prb_eval_list(actions)

These two procedures are explained above in connection with the
 [POP11 ...]
action type.

-- A format for defining prb_sortrules ------------------------------

Different search (conflict resolution) strategies can be defined for selecting between applicable rules by making prb_allrules true or 1, and defining the procedure prb_sortrules appropriately.

 All possible search strategies can be defined using a procedure prb_sortrules, of the following form, where prb_better is defined according to the preference strategy, as indicated below.

```
 define prb_sortrules(possibles) -> possibles;
 ;;; possibles is a list of applicable rule instances

 syssort(possibles, prb_better) -> possibles;

 if prb_allrules == 1 then
 ;;; truncate possibles list to length 1
 [] -> back(possibles)
 endif

 enddefine;
```

The procedure prb_better will be applied to two rule instances at a time, and can use the following procedures to access their components:

```
 prb_ruleof prb_varsof prb_valsof prb_foundof prb_recof;
```

-- . Selecting on the basis of specificity

This is one way to define prb_better so that it gives priority to rules with most conditions satisfied, would be to assign the following to it.

```
 define prb_more_specific(instance1, instance2);
 lvars instance1, instance2;
  listlength(prb_conditions(prb_ruleof(instance1)))
 >=
 listlength(prb_conditions(prb_ruleof(instance2)))  enddefine;
```

If the two rules have the same number of conditions, then the first rule found will come first.

-- . Sorting on the basis of the most recent condition made true

If one of the criteria for ordering possible rules is the relative recency with which their conditions have become true, then a crude approximation is provided to this if prb_recency is set true. In that case, as explained above, recency information will be recorded in rule instances, though for each item matching a rule condition the recency will be compared ONLY with other items in the same portion of the database, i.e. other items starting with the same first element.

In order to ensure that this recency information is recorded in the possibilities list, check that prb_max_conditions is big enough to accommodate as many conditions as each of your rules requires. (See its definition above.) Then do

```
 true -> prb_recency;
```

Then you could define the procedure for sorting rule-instances something like this. For each instance find its "youngest" matching database item, and then compare instances on the basis of how young their youngest items are.

```
 define prb_youngest(instance) -> num;
 ;;; Given a rule instance find the "age" of the most recently  ;;; added item making one of its conditions true.
```

```
 lvars n, instance, num=9999999;
 for n in_vector prb_recof(instance) do
 unless n == 0 then
 min(num, n) -> num
 endunless
 endfor
 enddefine;

 define prb_more_recent(instance1, instance2);
 ;;; given two rule instances return true if the first has the  ;;; most
recent enabling condition
 lvars instance1, instance2;
 prb_youngest(instance1)
 <=
 prb_youngest(instance2)
 enddefine;
```

Another possibility would be to replace prb_youngest with a procedure that computed the average "age" of the items.

A more complex rule could compare the two recency vectors until there is a difference, and if there isn't one then use specificity, for instance

```
 define prb_more_recent(instance1, instance2) -> boolean;
 ;;; given two rule instances return true if the first has the  ;;; most
recent enabling condition
 lvars instance1, instance2, vec1, vec2, num1, num2, boolean;

 define ages_list(vec) -> list;
 ;;; produce a sorted list of the non-zero numbers in vec  lvars vec,
list;
 [%appdata(vec,
 procedure (num); lvars num;
 unless num == 0 then num endunless
 endprocedure)%] -> list;
 sort(list) -> list
 enddefine;

 for num1, num2 in ages_list(vec1), ages_list(vec2) do

 if num1 > num2 then
 false -> boolean; return()
 elseif num2 < num2 then
 true -> boolean; return()  endif

 endfor;
 ;;; No difference in ages list, so compare specificity
prb_more_specific(instance1, instance2) -> boolean

 enddefine;
```

Naturally there is no uniquely best strategy: it is up to the user to define one.

NB: It will normally be better not to use prb_recency, but to set up your own way of recording relative age of database items. That can then be used as the basis for comparison. The above are given only as illustrative

examples.

-- Turning tracing of individual rules on or off ----------------------

If prb_walk is true, all rules will be traced. However, if it is false, information about individual selected rules can be provided, including interactive pauses before each action, by using the following facilities.

-- . prb_trace(<list of rule names>)
 Specifies that those rules should be traced. What this means is that the program will print out a message stating when the rule is being checked for applicability, and will indicate if its conditions are not satisfied. If they are satisfied and the rule is selected to be run, then during execution it will pause before each action, allowing the kind of interactive interrogation described above in the section on user interaction.

 To trace the checking of individual conditions use the variable prb_show_conditions.

-- . prb_untrace(<list of rule names>)
 This undoes the effect of prb_trace. However individual rules will still be stepped through if prb_walk is set true.

-- . prb_untrace("all")
 Unsets tracing on all rules.

-- Extended example: factorial --

This example has been moved to TEACH * POPRULEBASE

-- Extended example TEACH * PRBRIVER ---------------------------------

This teach file defines a production system that can solve the river-crossing problem described in TEACH * RIVER. It includes a complete solution that can be run once LIB POPRULEBASE has been compiled.

-- Autoloadable library procedures in $poplocal/local/newkit/prb/auto The following are autoloadable

-- -- prb/auto/prb_add_list_to_db.p
 prb_add_list_to_db(list, dbtable)

 add everything in the list to the database, not respecting order.

-- -- prb/auto/prb_allpresent.p

 prb_allpresent(patternlist) -> found;

 Check if all patterns in a list have consistent instances in prb_database (like * allpresent and * database).

-- -- prb/auto/prb_assoc_memb.p

 prb_assoc_memb(item, assoc_list) -> result

```
 given "b" and [[a] 1 [b c] 2 d 3] this returns 2
 given "d" and the above list it returns 3.
 given "e" and the above list it returns false.

 Compare prb_assoc, below. prb_assoc is faster, but does
 less checking.

-- -- prb/auto/prb_collect_values.p

 prb_collect_values(Spec) -> List;

This procedure takes a list which contains a pattern variable (indicated
by "?" followed by an identifier) followed by one or more patterns. It
finds all possible ways of consistently matching the patterns against
items in the database, and for each of them it remembers the value of the
identifier. It returns a list, possibly empty, of all the values. Repeated
values are pruned.

(Compare prb_which_values, described below).

Examples
 uses poprulebase;

 prb_newdatabase(16,
 [[joe isa man]
 [jill isa woman]
 [joe lives_in london]
 [jill lives_in brighton]
 [bill isa man]
 [sue isa woman]
 [bill lives_in london]
 [sue lives_in paris]]) -> prb_database;


vars x, town, personm, info;
prb_collect_values(! [?x [?x lives_in =]])==>
** [jill sue bill joe]

prb_collect_values(! [?town [= lives_in ?town]])==>
** [brighton paris london]

prb_collect_values(! [?info [??info london]])==>
** [[joe lives_in] [bill lives_in]]
 prb_collect_values(![?person [?person isa woman][?person lives_in
paris]])==> ** [sue]

prb_collect_values(
 ![?person [?person isa woman][NOT ?person lives_in paris]])==>
** [jill]

See also LIB * PRB_COLLECT_VALUES, HELP * PRB_COLLECT_VALUES


-- -- prb/auto/prb_delete_rule.p
 prb_delete_rule(word, ruletype)
 Remove a rule, named by word, from the list of rules named
 by ruletype
```

```
-- -- prb/auto/prb_do_all.p

 prb_do_all(rule_instance, actions);
 ;;; actions should be of form [DOALL action action action ]

-- -- prb/auto/prb_foreach.p

 prb_foreach(pattern, proc);

 Apply proc for every match between pattern and a database item  (Compare
* foreach and * database)

-- -- prb/auto/prb_implies.p

 prb_implies(patternlist, pattern) -> boole;

 Interprets [IMPLIES <patternlist> <pattern>] conditions

-- -- prb/auto/prb_interact.p

 prb_interact(message, action, rule_instance, accept_empty) -> answer;

 Handles interaction in actions, e.g. menus.  [Documentation needs to be
expanded]

-- -- prb/auto/prb_list_data.p
 procedure prb_list_data( dbtable ) -> list

 return a list of all the data items in the database.


-- -- prb/auto/prb_make_rule.p

 prb_make_rule(list);

 Needed for actions of type [RULE ...], described above

-- -- prb/auto/prb_map_action.p

 prb_map_action(rule_instance, action);

 Invoked by actions of the form
 [MAP ?veclist <procedure> <action1> <action2> ... <actionn>]
 The procedure must take a veclist and an actionlist
 and decide what to do about actions in the actionlist on the  basis of
items in veclist.

-- -- prb/auto/prb_menu_interact.p
 defined in prb_read_info

 define procedure prb_menu_interact =
 $-prb$-prb_read_and_add(%true%)
 enddefine;

-- -- prb/auto/prb_pause_read.p
 defined in prb_read_info
```

```
 define prb_pause_read(rule_instance, action);
 lvars rule_instance, action;
 if prb_pausing then  ;;; wait for user to press return
prb_read_and_add(rule_instance, [READ '' [==] []], false)  endif
 enddefine;


-- -- prb/auto/prb_pr_rule.p

 prb_pr_rule(rule);
 can have an optional list as second argument
 Prints a rule so that it can be re-compiled.

-- -- prb/auto/prb_print_menu.p

 prb_print_menu(question, options);
 For MENU actions.
 A list of question items and a list of options

-- -- prb/auto/prb_push_or_pop.p

 prb_push_or_pop(list, pushing);
 Used by [PUSH...] and [POP...] action types

-- -- prb/auto/prb_read_info.p
This file defines several procedures

 prb_read_and_add(rule_instance, action, with_menu);

 Handles the "READ" and "MENU" actions.

 define procedure prb_read_info =
 $-prb$-prb_read_and_add(%false%)
 enddefine;


-- -- prb/auto/prb_remove_all.p
 prb_remove_all(list_of_patterns)-> found;

-- -- prb/auto/prb_replace.p
 prb_replace(item, value, list) -> list;
  Replace all occurrences of item (at any depth) with contents of  value
in a copy of list. If value is a list, its elements are  spliced in.
 E.g.

 vars ANSWER = [age 27];
 prb_replace("ANSWER", ANSWER,
 [information [new_value ANSWER] ]) =>

 ** [information [new_value age 27]]

 This is analogous to the use of "^^ANSWER", except that the value of
ANSWER need not be a list.

-- -- prb/auto/prb_rule_weight.p

 prb_rule_weight(rulename, /*type*/) -> weight;
```

```
 weight -> prb_rule_weight(rulename, /*type*/)

 Can have an optional list of rules as second argument.
 Gets or updates the weight associated with a rule.

-- -- prb_ruleschanged_trace(ruleset, family);

User definable procedure. May run if prb_trace_ruleschanged is true and
the current ruleset changes. It is run before the ruleset gets
interpreted. Does nothing by default.

-- -- prb/auto/prb_save_or_restore.p

 prb_save_or_restore(action, list);  For [SAVE ...] or [RESTORE ...]
actions


-- -- prb/auto/prb_saverules.p

prb_saverules(<ruleset>, <rulesetname>, <filename>)

This is intended only for saving relatively simple rulesets in a file.

Saves the ruleset given with the rulesetname in the specified file.
Will overwrite any existing contents in the file.

Does not handle conditions and actions that are capable of including Pop-
11 code, e.g.

 [VARS ...]
 [LVARS ...]
 [WHERE ...]
 [POP11 ...]

Unfortunately it is impossible to decompile the code in these
structures in order to determine what to save.


-- -- prb/auto/prb_select_action.p

 prb_select_action(rule_instance, action);
 Invoked by actions of the form
 [SELECT ?veclist <action1> <action2> ... <actionn>]
 where the veclist was produced by a filter condition

-- -- prb/auto/prb_show_rules.p

 prb_show_rules(rule_list);
 Display all rules in the list

-- -- prb/auto/prb_trace.p

 define prb_trace(list);
 lvars word, list;  for word in list do
 true -> prb_istraced(word)
 endfor;
 enddefine;
```

-- -- prb/auto/prb_trace_rule.p

 prb_trace_rule(rule_instance, instantiate);
 Print out a rule instance
 If instantiate is true then instantiate it first.

-- -- prb/auto/prb_truncate.p

 prb_truncate(list,num) -> list;
 Truncate list to length num
 Used to shorten memory in poprulebase

-- -- prb/auto/prb_untrace.p

 prb_untrace(list);
 To turn off tracing of types in list. If list is "all" turn
 off all tracing.

-- -- prb/auto/prb_walk_trace.p

 prb_walk_trace(rule_instance, action);
 Calls prb_interact to print out which rule is currently
 being invoked and which action in that rule.

-- -- prb/auto/prb_which_values.p

prb_which_values(Vars, Patternlist) -> List;

The procedure prb_which_values, is a replacement for "which_values", as explained in HELP * WHICH_VALUES, which does not work with Poprulebase.
 It takes a list of pattern variables, each indicated by "?" followed by a word or identifier, and a list of patterns. It tries all possible ways of consistently matching the patterns in Patternlist against items in the database (prb_database), and for each of them it makes a list of the values of the variables in the Vars list. All such lists are then put into a single large list, in the order in which they are found, and that list is returned as the value of prb_which_values.
 See HELP * PRB_WHICH_VALUES

This can be compared with the somewhat more simple prb_collect_values, which can handle only one variable at a time.

Examples:

 uses poprulebase;

 prb_newdatabase(16,
 [[joe isa man]
 [jill isa woman]
 [joe lives_in london]
 [jill lives_in brighton]
 [bill isa man]
 [sue isa woman]
 [bill lives_in london]
 [sue lives_in paris]]) -> prb_database;

vars x,y; prb_which_values(! [?x ?y], ! [[?x lives_in ?y]])==>

```
** [[jill brighton] [sue paris] [bill london] [joe london]]

prb_which_values(![?person], ! [[?person isa woman]])==>
** [[jill] [sue]]

prb_which_values(![?person],
 ! [[?person isa woman][NOT ?person lives_in paris]])==>
 ** [[jill]]
```

-- Utility procedures

Some of these have their own library files. Others can be examined using the "ENTER showlib poprulebase" command and searching for the procedure definition. (See HELP ved_headers, available as a Birmingham Poplog extra.)

-- . prb_new_rule(<name>, <weight> <conditions>, <actions>, <type>)

This is the main procedure for constructing rules. Users will not normally invoke it directly, but might do so if they wish to change the syntax used for rule definitions.

<name> is a word, the name of the rule
<weight> is a number, ignored if prb_useweights is false <conditions> is a list of conditions, i.e. a list of lists
<actions> is a list of actions, i.e. a list of lists <type> is a word, the name of the ruleset to be used

Given a word, a weight, a list of conditions, a list of actions, and a word that names a ruleset, the procedure prb_new_rule does exactly the same as the "define :rule" form described below, i.e. it updates an existing rule in prb_rules, or creates a new one at the end.

It also does some checks to ensure that the arguments are of the correct type (e.g. weight should be a number). It probably does not do enough checks, so should be used with care.

See also the [RULE ...] action format, which can also be used for adding rules at runtime.

-- . prb_delete_rule(<name>, <type>)

This removes the rule with the given name from the list specified by the word <type>.

<name> should be the name of a rule that is already in the list that is the value of the word <type>.

If the location of a rule in the list is to be changed, the rule must first be removed from the list using prb_delete_rule.

This can be done in an action of a rule, e.g.
  [POP11 prb_delete_rule(name)]

or less efficiently thus:

```
 [NULL [apply prb_delete_rule ?name]]
or
 [NULL [$: prb_delete_rule ?name]]
```

(See explanation of NULL actions, "popval" elements and "apply" elements, below).

It would also be possible to define a new action type of the form

```
 [DELETERULE ?name]
```

as shown below in the section on "User-defined action keywords".

```
-- . prb_pr_rule(<name>, <list>)
or
-- . prb_pr_rule(<name>)
```

The second, optional, argument, <list>, should be a list of rules. If it is not provided it defaults to prb_rules.

This procedure gets the rule of that name from the list and prints it out in a format that allows it to be recompiled.

```
-- . prb_forget_rules();
```

System utility, used to clear memory of remembered rule instances, e.g. when changing rulesets

```
-- . prb_assoc(key, list) -> val;
```

List is of form [key val key val key val ....]
Returns val corresponding to key in list, or false

```
prb_assoc("b", [a 1 b 2 c 3]) =>
** 2
prb_assoc("d", [a 1 b 2 c 3]) =>
** <false>
```

Compare prb_assoc_memb, above, which is more general and does more checking.

```
-- . prb_database_keys(dbtable) -> keys;
```

get the keys currently used in the table

```
-- . prb_is_var_key(key);
```

Key is the first item of a pattern, normally a word.
Check if it indicates that the pattern starts with a variable

```
-- . prb_in_data(pattern, data) -> item;
```

Return first item in the list data that matches pattern, otherwise false. Assume data is a list, so use fast_for
NB Does not localise popmatchvars

```
-- . prb_member(item, list) -> boolean
```

Default value is fast_lmember. Can be changed to lmember
if needed for debugging.

-- . sys_print_ident(id)

This procedure is assigned to class_print(ident_key), and is
used to print out identifiers. By default it calls the procedure
print_ident, which is defined thus in poprulebase, to help
ensure that when patterns and rule-conditions are printed, out
the printing is more helpful than the default print format for
pop-11 identifiers. The main change is that where the identifier has been
used in a poprulebase pattern the associated word is
remembered in the property word_of_ident, so that word
associated with the identifier can be included in the print out,
as well as the idval/valof. This is essential for debugging.

```
define vars print_ident(id);
 dlocal pop_pr_level = 3, pop_oc_print_level = 1;

 lvars word;
 if isproperty(word_of_ident) and (word_of_ident(id)->>word) then
printf('<ID %P %P>', [%word, idval(id)%])
 else
 printf('<ident %P>', [%idval(id)%])
 endif
enddefine;
```

-- . prb_variables_in(list, varlist, VARlist) -> (varlist, VARlist);
System procedure. Use showlib poprulebase to see details.

-- . prb_extend_popmatchvars(list, matchvars) -> matchvars;
System procedure, used to extend the list of variables whose names
can be used in patterns with pattern prefix.

-- . prb_valof(word) -> item;

System utility used for controlled dereferencing. Does not recurse, like
recursive_valof. Also resets popautolist to ensure that normal autoloading
works.

-- Additional procedures defined in LIB * POPRULEBASE -----------------

```
 define vars procedure isprb_rulefamily(x);
 define vars $-prb_instance(Pattern) -> value;
 define vars $-prb_value(Pattern) -> Pattern;
 define global vars procedure prb_condition_type
 define global procedure read_in_items();
 define global procedure read_list_of_items() -> result;
 define constant procedure prb_declare(word);
 define constant procedure prb_ldeclare(word);
 define vars prb_declare_lvar(item);
 define prb_read_VARS(usevector, name, lexical) -> varspec;
 define vars procedure prb_readcondition(vars_spec) -> condition;
 define vars procedure prb_readaction(vars_spec) -> action;
 define prb_read_conditions(vars_spec) -> (list, item);
 define prb_read_actions(vars_spec, terminators) -> (list, item);
```

```
 define prb_extract_vars(conditions, actions) -> rulevars ;
 define init_prb_rule(name, weight, conditions, actions, type, rulevars,
create);  define vars prb_new_rule = init_prb_rule(%false%)  define global
vars procedure prb_istraced =
 define vars procedure prb_applicable(rules) -> possibles;
 define prb_do_DEL(item, foundlist);
 define prb_DEL(rule_instance, action);
 define prb_instance_present(list) /* -> boolean */;
 define prb_ADDIF(rule_instance, action);
 define prb_ADDUNLESS(rule_instance, action);
 define vars procedure prb_do_action(action, current_rule, rule_instance);
define vars prb_do_action_now(list);
 define prb_do_in_data(rule_instance, action);
 define procedure prb_do_rules(rules);
 define global vars procedure prb_no_rule_found_trace(rules, data);
define global vars procedure prb_no_rule_found_action(rules, data, cycle);

-- Debugging and development aids ---------------------------------

-- . Use of indirection through identifiers

One feature that supports development and increases run-time flexibility
is the use of identifiers that refer to various entities used in an
application of poprulebase, e.g. the rulesets and rulefamilies. This means
that often a definition, e.g. of a ruleset, can be recompiled at run time
and the new version of a ruleset will then be available immediately
without having to restart the program.

This also allows a ruleset to be changed temporarily to turn tracing on
then off.

These features help to speed up program development and rapid prototyping.
If pop_debugging is made false, some of the references that go via
identifiers in the pop-11 dictionary are replaced by direct references to
the ruleset or rulefamily.

See also
 prb_trace_ruleschanged
 prb_ruleschanged_trace(ruleset, family)

Further information about flexibility and support for programs that
understand themselves can be found in the documentation on the sim_agent
toolkit:
 HELP SIM_AGENT
 HELP NEWKIT
 HELP RULESYSTEMS



-- . prb_profile

prb_profile is an extension of the main LIB PROFILE, described in
HELP PROFILE.

To make it available do:

 uses prb_profile
```

Then in order to use it all you have to do is run poprulebase or sim_agent (or any other program using poprulebase) with the command form

```
 prb_profile <command>
```

E.g. this was tested on a program written by Ian Wright as follows:

```
 ;;; Compile the library
 uses prb_profile

 ;;; Run command "go(200)" with the profiler recording procedure
 ;;; calls and activations of rules.
 profile go(200);
```

At the end it printed out the following:

```
;; CPU Time taken: 135.58 seconds. Number of interrupts: 12220
;
;;; PERCENTAGES OF TOTAL TIME:-
 38.81 sysmatch
 6.60 prb_forevery_sub
 5.01 sim_run_agent(object:sim_object,objects)
 3.76 get_intersection(line1:Line,line2:Line)
 3.58 isindata
 3.08 sim_distance(a1:sim_object,a2:trsim_bar_agent)
 2.47 draw_agent(a:trsim_agent)
 1.57 collision_detect(x1,y1,x2,y2,polygon:poly)
 1.53 prb_value
 1.44 return_intersections(x1,y1,x2,y2,polygon:poly)
;;; Number of times rules active: 7095
;;; PERCENTAGES OF TOTAL:-
 60.86 belief_maintenance_r1  6.29 update_beliefs
 5.82 grule_low_charge_r1
 3.92 TR_amble_r2
 3.59 ditch_kill_bug
 2.02 create_new_belief
 1.92 TR_amble_r3
 1.00 TR_make_wall_r2
 0.70 TR_goto_r2
 0.69 runTRP
```

The first part of this is the standard printout that is produced by the Pop-11 profile as described in HELP * PROFILE.

The second part is specific to prb_profile. It informs us that the rule interpreter was active approximately 7095/12220 = 0.58 of the time and of that 60% of that time is spent in the belief_maintenance rule.

Similarly, running the code in TEACH * SIM_DEMO (in the SIM_AGENT library) in a particular configuration, with graphics turned on (on a Sparcstation IPC, with SunOS 4.1.3 and Poplog version 15.01) gave the following (after 200 cycles of the scheduler):

```
 ;;; CPU Time taken: 124.83 seconds. Number of interrupts: 10829
 ;;; PERCENTAGES OF TOTAL TIME:-
 12.36 prb_forevery_sub
 8.06 sim_run_agent(object:sim_object,objects)  5.57 prb_value
```

```
 5.52 isindata
 4.49 prb_do_action
 3.76 record_rule_instance
 3.76 sysmatch
 3.58 prb_run_with_matchvars
 2.58 prb_flush
 2.51 prb_applicable
 ;;; Number of times rules active: 7219
 ;;; PERCENTAGES OF TOTAL:-
 26.04 team_process_obstacle
 23.17 see_something
 18.01 team_process_percept
 5.36 team_move_to
 4.68 see_obstacle
 3.84 sim_check_at1
 3.44 see_target1
 1.79 see_finish_processing
 1.77 team_no_percept
 1.57 see_nothing
```

This informs us that the rule interpreter was active approximately 7219/ 10829 = 0.67 of the time and that 26% of that time is spent in the the team_process_obstacle rule.

(NB: the SIM_DEMO file may have changed since the above. Also computers have speeded up considerably, causing profiler output to change.).

-- . LIB prb_trace_procs (and show_trace_match)

To permit extended tracing of rules and conditions, do
 uses prb_trace_procs
 after setting up poprulebase search lists but before compiling the main poprulebase library, as described in HELP PRB_TRACE_PROCS

This extension to poprulebase gives more detailed information about which conditions are tested most often, the proportions of times they are satisfied, which agents use them, etc.

After a ruleset has been running for some time, the procedure show_trace_match is used to print out information about occurrences of matching in particular rules.

The global variables and procedures defined there are
show_trace_match(false);
 show_trace_match(rulename);
 show_trace_match(list);
 prb_trace_rule_stats(ruleinfo, name) -> stats
 prb_rules_traced() -> names
 prb_sort_traces(names, num) -> sorted
 prb_show_full_trace(num)
 prb_object_matches(object_name) -> list
 clear_trace_match()

-- . LIB prb_checkpatterns

prb_checkpatterns(infile, oklist, outfile);
prb_checkpatterns(infile, oklist, outfile, true);
 This procedure defined in LIB PRB_CHECKPATTERNS makes it much easier to track down errors due to mis-typing something in a rule condition or action.

The procedure can be used to create a file of information (outfile) about keywords and other symbols used in conditions and actions in a file defining rulesets (infile), excluding symbols defined by
the list oklist. The optional fourth argument suppresses line numbers and duplicate occurrences.

For details see
 HELP PRB_CHECKPATTERNS

-- Note on efficiency --

In the implementation of POPRULEBASE, clarity, maintainability, and functionality have been given higher priority than efficiency. For example see the section above on use of indirection.

If necessary a more restrictive version may be provided later which gives higher priority to efficiency. In particular, sophisticated indexing schemes are possible which are not used in this package as they would add significantly to the complexity. Moreover, by partitioning rulesets and databases so that at any time only fairly short lists are involved, users may be able to achieve the same effect as clever indexing far more simply.

The global variable prb_copy_modify illustrates one of the devices for increasing efficiency, which is potentially dangerous.

-- Further Reading ---

See standard text books on AI and Expert Systems. The following are online files accessible from the Poplog editor Ved.

-- . Introductory documentation
 (Some of these teach files may not be available outside Birmingham)

 TEACH * RULEBASE
 Introductory overview
 TEACH * POPRULEBASE
 Gives more advanced examples
 TEACH * PRBRIVER
 Described above - a rule-based planning program for solving
 the river crossing problem.
 TEACH * PRBWINE
 Describes a simple wine adviser expert system
 TEACH * PRBZOO Describes a simple animal classifier based on a
 discrimination net
 TEACH * PRBGROCERIES
 An expert system for packing groceries into bags at a supermarket checkout.

Note: some of the above files may still use the older "define :rule..."

syntax instead of "define :ruleset...". This is described in

HELP * OLDRULESYNTAX

-- . Other teach files on expert systems and rule-based systems:

TEACH * EXPERTS - an introduction to expert systems and expert system shells

TEACH * PSYS - a very primitive production system interpreter

TEACH * PRODSYS - a more complex one, though not as flexible is LIB POPRULEBASE.

-- . More advanced features of Poprulebase

HELP * RULESYSTEMS
Describes rulesets, rulefamilies and rulesystems, and other features relevant to the sim_agent toolkit.

HELP * PRB_EXTRA
Some additional action types for switching rulesets or databases
HELP * PRB_FILTER
Additional condition types and action types for interfacing rules with neural nets and other mechanisms.

HELP * PRB_TRACE_PROCS
Describes a facility for keeping records of which matches are attempted during rule testing. Subsequently information of various kinds about numbers of matches tried and failed in various rules or in connection with various agents (in the Sim_agent toolkit) can be printed out to help with debugging or improving efficiency of programs.

HELP * PRB_NEWS
Describes recent changes to the system.

HELP * VED_FIXRULE
Describes a Ved-based utility for converting rules in the old "define :rule..." format to the new "define :ruleset" format.

-- . The use of Poprulebase in the Sim_agent toolkit
(If LIB *SIM_AGENT is installed)
TEACH * SIM_AGENT
TEACH * SIM_DEMO
HELP * SIM_AGENT

Further features are described in comments in the program library file. To examine it
SHOWLIB * POPRULEBASE

-- PRB_EXTRA is now redundant ----------

Before the introduction of rulefamilies, there was an extension to poprulebase, called LIB PRB_EXTRA, described in HELP * PRB_EXTRA. The

facilities in that library support stack mechanisms for temporarily changing the current database or the current set of rules. These are now superseded by the rulefamily mechanisms described in HELP * RULESYSTEMS. If you need to use the older mechanisms, you can make them available with the command:

```
uses prb_extra
```

(Not to be used with the SIM_AGENT library).

-- Acknowledgements ----------

Some of the key ideas concerned with variable binding and the Pop-11 pattern matcher come from early work on Pop-11 by Steve Hardy. Subsequent work on rule-based systems in Pop-11 by Steve Hardy, Chris Mellish, Allan Ramsay, Mike Sharples and Tom Khabaza all contributed to the ideas in this package as implemented originally in LIB NEWPSYS
(now obsolete).

Many of the features in described in HELP * PRB_FILTER arose out of discussions with Riccardo Poli, whose experiments using the system also helped to iron out some problems.

The idea of adding weights, and the first implementation of the weights, came from Tim Read.

Darryl Davis helped with conversion of the package to use property tables for the database instead of a list of lists.

Jeremy Baxter at DRA Malvern made many comments and suggestions as a result of using early versions, and finding efficiency bottlenecks.

Brian Logan suggested adding [DLOCAL ..] forms, by analogy with Pop-11 dlocal expressions. (See HELP * DLOCAL).

Several suggestions have come from other users, including Catriona Kennedy, Nick Hawes, and Matthias Scheutz.

--- $poplocal/local/newkit/prb/help/poprulebase