5

Environment

IN THIS CHAPTER you will find a description of the command-line options, of the compilation process, of the Prolog IV debugger, and of some other facilities which are here to help the programmer.

5.1 Command-line options

Prolog IV can be launched from a shell (xterm, cmdtool) without its graphical environment. This is done by using the following command

```
% prolog4 [ option ... option] [-- userOption...]
```

for which the options are described below, and the user options may be any string. The bracketed items are optional, and the double-dash is used to separate the options to be interpreted by Prolog IV from the options which are to be interpreted by a Prolog IV program.

All options can be retrieved from a Prolog IV program by the `argv/1` primitive as a list of identifiers. The primitive `user_argv/1` may also be used to retrieve only the user part of the options (*i.e.* the options which come after the double dash).

5.1.1 Options

Most of the command-line options concern the settings of the various stacks and workspaces used by Prolog IV (in particular concerning their size). Some other options may be used which affect the default settings of Prolog IV.

These options may be given as command-line options (as described above), or through a preferences dialog in Prolog IV's graphical interface.

In the following, we call *cell* a doubleword (*i.e.* eight bytes on a 32-bit machine).

The following list gives a few of these options. The default value is given between brackets. In all cases, Nb is a positive number which must be provided as parameter to the option.

`-help`	Displays the list of available options.
`-heap` Nb	Sets the size of the *heap* stack to Nb cells [700000]
`-trail` Nb	Sets the size of the *trail* stack to Nb cells [400000]
`-env` Nb	Sets the size of the *env* stack to Nb cells [50000].
`-choice` Nb	Sets the size of the *choice* stack to Nb cells [50000].
`-global` Nb	Sets the size of the *global* workspace (consult, record) to Nb cells [500000].
`-local` Nb	Sets the size of the *local* stack (unification, constraint solving) to Nb cells [5000].
`-work` Nb	Sets the size of the *temporary* zone (precise computations, linear solver) to Nb cells [50000].
`-nbpcindex` Nb	Sets the maximum number of litterals in compiled programs (pCode) [10000].
`-union`	Starts Prolog IV in « unions » mode [« simple »].
`-iso`	Starts Prolog IV in « iso » syntactical mode [prolog4].
`-g`	Starts Prolog IV in « debug » mode [no_debug].
`-Qcalculator=off`	Starts Prolog IV without the rational calculator [on].

5.1.2 Primitives

argv(L), user_argv(L) are used to retrieve the command line arguments, as a list of atoms. **user_argv/1** only considers the arguments which occur after the separator `--` ; it returns `[ ]` if there is none. For instance:

```
machine% prolog4 -heap 30000 -- carres 9
...
>> argv(L).
L = ['prolog4','-heap','30000',--,carres,'9'].
>> user_argv(L).
L = [carres,'9'].
```

5.2 User interrupt

A Prolog IV program may be interrupted at any time by using a predefined key, combination of keys or another action, which depends on the underlying operating system and environment. For instance, the combination `CTRL-C` is used under Unix, or the « Stop » button in Prolog IV's graphical console.

The program interruption may be used for two reasons:

- Stop a running program (either because it is in a loop or because it simply ran for too long).
- Switch to another execution mode (in debug mode).

Under Unix, the same key is used for both purposes. The discrimination is made through the way in which the key is used:

- If the key is struck *twice* in a second or less, then Prolog IV switches to debug mode, and gives control to the user (at the debugger prompt).
- If the key is ony struck once, it is interpreted as a program interrupt request. If this error is not handled by the program, it terminates, and the user gets back to the prompt.

5.3 Compiling rules

After recalling briefly the definition of the rule input primitives, we will give a description of the directives which may be used within a compiled file.

5.3.1 Rule input primitives

There are two familiies of rule input primitives:

- those which encode rules as dynamic rules (**consult** family),
- those which encode rules as static rules (**consult** family).

Most of the time, you will rather compile rules than consult them: the execution of compiles rules is faster, and the debugger only works with compiled rules.

compile(F**), compile(**F**,** L**)** compile the file named F (an atom). Compiled rules are static. L is a list of options about the syntactical mode used and the debugging mode for that particular compilation.

recompile(F**), recompile(**F**,** L**)** work like `compile/`n, but allow the user to redefine predicates.

consult(F**), consult** read and encode the file named F (an atom). Consulted rules are dynamic. The version without arguments reads rules from the current input.

reconsult(F**), reconsult** work like `consult/`n, but allow the user to redefine predicates.

5.3.2 compiler directives

Directives are annotations which are used to provide the compiler with some information, to alter its behaviour, to specify some properties of rule blocks, to include files, ... Directives are only understood by the compiler, and they are ignored when a file is consulted.

The list of these directives follows:

dynamic/1 Behaves like the primitive `dynamic/1`. In the present version of Prolog IV, the compiler is only able to compile, so the presence of such a directive will trigger an error condition.

op/3 Behaves like the **op/3** primitive. In particular, the operators declared remain valid after the end of the compilation.

syntax/1 Informs the compiler that the rules blocks which follow the directive are to be compiled in mode

- `iso`, when the major syntactical extensions should be deconnected,
- `prolog4`, when the extensions which make it easier to write constraints should be connected.

The default mode is indicated on the console, where the prompt is >> in `prolog4` mode, and `?-` in `iso` mode. When starting Prolog IV, the default mode is `prolog4`.

include/1 The directive `:-include(`F`)` tells the compiler to read the file F (an atom) before to proceed with the compilation of the current file. The compiler behaves just as if the directive was replaced by the contents of the file F.

debug/0 The rule blocks which follow this directive (until the end of the file or an occurrence of the `:-no_debug` directive, should be compiled in debug mode, in order to enable the use of the debugger on these blocks.

no_debug/0 The rule blocks which follow this directive (until the end of the file or an occurrence of the `:-debug` directive, should not be compiled in debug mode; the debugger will not be active on these blocks.

set_prolog_flag/2 Simply calls at this point the primitive `set_prolog_flag/2`.

5.4 The Prolog IV debugger

5.4.1 Introduction

Any program (even those written in Prolog IV) can be wrong!
There is no programming language which allows its users to enter in the machine « the program they think about » without leaving them the opportunity to add bugs to this program. Of course, one can look at the code which does not work, and may find some bugs this way. Furthermore, one should pay attention to the warnings issued during the compilation of the file or its syntactic checking; more bugs will be found this way. However, in many cases, this is not enough, and a detailed examination of the program execution using an interactive tool may lead to the discovery of many more errors. The Prolog IV debugger has been designed to meet that need.

5.4.2 The box model

The Prolog IV debugger is based on the « box model » which has also been used in many commercial implementations of Prolog. The key concept in this model is *every predicate call is considered as a box*, which may be a glass-box or a black-box, depending on the user's need to have detailed traces or not. The boxes in this model have four openings, which are called *ports*. Two more ports are available in Prolog IV, which allow us to peek inside the boxes.

5.4.3 The ports

The ports correspond to precise spots in the execution of a goal, namely:

- the activation of a rule block (`CALL`),
- the termination of one of the block's rules (`EXIT`),
- a backtracking inside the block (`REDO`),
- the end of a block's activation, by elimination of all its rules (`FAIL`),
- the selection of the n-th rule of a block (`RULE`),
- the end of the head unification with the current goal (`OK`).

The two last ports are secondary ones, which are inside the box.

The behaviour of a box is naturally non-deterministic, and the same box may be solicited several times in order to give several different answers. Sequentially, a box is entered via a `CALL` port. It is then exited via the `EXIT` port when it succeeds (a solution has been found). When choices have been left hanging, the box is reentered through the `REDO` port. It may be exited again through the `EXIT` port if a solution is found. When there are no more solutions available, the `FAIL` port is used to exit definitely.

The display of ports respects the same schema, like in the following example:

```
4 [2] EXIT (r1) : dessert(glace)
```

Here is the description of the various fields used here:

`4 :` Identification of the call : This is the running number of the box from from which the call has been performed.

`[2] :` Depth of the call : This is the leel of imbrication of the box (*i.e.* the number of calling boxes around the current box). This number is of course the same for all the calls found in the same rule tail (or query).

`EXIT :` This is the name of the port, *i.e.* one of the strings `CALL`, `EXIT`, `REDO`, `FAIL`, `RULE` ou `OK`.

`(r1) :` The number of the rule applied. When followed by a star, it shows that this rule is the last in the block.

`dessert(glace) :` The litteral currently being handled. It is displayed in its current instantiation state.

The ports are displayed just before to be traversed.

5.4.4 Debugging sessions

We will now look at a few examples of debugging sessions, in which we will see the various capacities of the Prolog IV debugger, using the most common commands. In this session, all that needs to be typed by the programmer is underlined.

Let's consider the program below, which we place in the `dessert.p4` file:

```
dessert(fruit, 2) .
dessert(ice_cream,6) .
```

Let's compile this file in `debug` mode:

```
>> compile('dessert.p4', [debug(on)]).
true.
```

Let's now follow the execution of a call to the `dessert` predicate; every time that the debugger will display its prompt, we will type the command `s`, a shortcut for `step`, which goes from port to port.

In order to simplify the presentation, somme line feeds have been removed, as well as some text in the presentation of solutions.

```
>> debug.
true.

>> dessert(X,Y).

   4[4]CALL    : dessert(_267, _268) (DBG IV) s
    5[5]RULE(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r1): dessert/2(fruit, 2) (DBG IV) s
    5[5]EXIT(r1): dessert/2(fruit, 2) (DBG IV) s
Y = 2,
X = fruit
  4[3]REDO(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]RULE(r2*): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r2*): dessert/2(ice_cream, 6) (DBG IV) s
    5[5]EXIT(r2*): dessert/2(ice_cream, 6) (DBG IV) s
Y = 6,
X = ice_cream
    5[5]FAIL(r2*): dessert/2(_267, _268) (DBG IV) s

>>
```

Here is now a step-by-step explanation of the session above:

- (`CALL`) We are calling `dessert`.
- (`RULE`) The first rule (`r1`) of `dessert` is selected.
- (`OK`) The rule head is unified with the call.
- (`EXIT`) All the tail literals have succeeded (they were none).
- The solution is displayed, since the execution queue is empty.
- (`REDO`) The exists a choice point for the call to `dessert`.
- (`RULE`) We select the second rule (`r2`) of `dessert`.
- (`OK`) The rule head is unified with the call.
- (`EXIT`) All the tail litterals have succeeded (they were none).
- The solution is displayed, since the execution queue is empty.
- (`FAIL`) There are no more choice points in `dessert`.
- We are back to the Prolog IV prompt. The whole search tree has been explored.

In order to go a little further in our exploration of Prolog IV's debugger, let's now use the file `menu.p4` below:

```
repasleger(H,M,D) :-
    horsdoeuvre(H,I),
    plat(M,J),
    dessert(D,K).

horsdoeuvre(radis,1) .
horsdoeuvre(melon,6) .

plat(M,I) :- viande(M,I) .
plat(M,I) :- poisson(M,I) .

dessert(fruit, 2) .
dessert(glace,6) .

viande(poulet,5)  .
viande(porc,7)  .

poisson(sole,2) .
poisson(thon,4) .
```

Let's see in this session how we can move from box to box without looking at the details of a call.

```
>> compile('menu.p4', [debug(on)]).
true.

>> debug.
true.

>> repasleger(X,Y,Z).

    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) s
     3[3]RULE(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]OK(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]CALL    : horsdoeuvre(_267, _1213) (DBG IV) n
     3[3]CALL    : plat(_268, _1214) (DBG IV) n
     3[3]CALL    : dessert(_269, _1215) (DBG IV) s
      7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV)
     ...
```

- For the three first ports, we have proceeded step-by-step using the `step` command.
- We want to skip the details of the execution of the `horsdoeuvre` and `plat` predicate. We then use the `next` command (whose shortcut is `n`).
- When reaching the call to `dessert`, we get back to the step-by-step mode, in order to see the details of the execution.

Let's now see how we can move even faster (still without looking at the details), by using breakpoints.

```
>> debug.
true.

>> repasleger(X,Y,Z).
    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) spy dessert

   dbg: spying: (dessert/2).
   (DBG IV) c
   **   3[3]CALL    : dessert(_269, _1215) (DBG IV) s
   **    7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]OK(r1): dessert/2(fruit, 2) (DBG IV) c
   **    7[4]EXIT(r1): dessert/2(fruit, 2) (DBG IV) c
   Z = fruit,
   Y = poulet,
   X = radis
   **  6[2]REDO(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]RULE(r2*): dessert/2(_269, _1215) s
   **    7[4]OK(r2*): dessert/2(glace, 6)
   ...
```

- We use the `spy` command in order to set a breakpoint on the rules in the `dessert` block.
- Next, we use the command `cont` (whose shortcut is `c`) in order to proceed with the execution until the program ends or a breakpoint is reached.
- When the machine stopsm we then use `step` in order to follow the execution of `dessert`.

The `**` markers in the left margin show that the port is a breakpoint.

Let's now show some commands which display various information about the state of the Prolog IV machine (related to the ongoing demonstration) and about the active rule's variables:

```
(DBG IV) where
---\/---  NEW  ---\/---
   7[3](r2*) dessert/2(glace, 6)
   3[2](r1*) repasleger/3(radis, poulet, glace)
   2[1](r1*) sysh__query/1(['X'=radis,'Y'=poulet,'Z'=glace])
---/\---  OLD  ---/\---
(DBG IV) chpt
---\/---  NEW  ---\/---
   6[4](r1) viande/2(poulet, 5)
   5[3](r1) plat/2(poulet, 5)
   4[3](r1) horsdoeuvre/2(radis, 1)
---/\---  OLD  ---/\---
(DBG IV) up
(DBG IV) printvar
H = radis
M = poulet
D = glace
I = 1
J = 5
K = 6
(DBG IV)
```

- The `where` command is used to see where we are in a demonstration, *i.e.* which are the active rules.
- The `chpt` command is used to see the choice points in which there still are available choices (*i.e.* the active choice points).
- The `up` command sets the calling rule as active procedure. This allows us to have a look at the values of the variables at this level.
- The `printvar` is used to do just that, *i.e.* to print the current values of all active variables in the active rule.

5.4.5 Debugger options

The options[1] are the debugger's internal variables. They are mostly used in order as implicit parameters to some of the debugger commands.

In many command and options, the ports are identified by a letter. We have *in extenso* `c` for `CALL`, `e` for `EXIT`, `r` for `REDO`, `f` for `FAIL`, `R` for `RULE` and `Y` for `OK`. A combination of ports is a sequence of these letters, such as « `ceR` ». An empty combination is expressed by « `{}` ».

Such sequences are used in the debugger options `stepports`, `nextports`, `contports`, `spyports`, which are described below.

stepports : is used by the `step` command to know on which ports the `step` command should stop. By default, this option contains the string « cerfRY » (*i.e.* all ports).

nextports : is used by the `next` command to know which ports should be displayed during a `next` command. By default, this option contains the empty

[1] The word « option » is used in order to avoid any confusion with Prolog variables.

string (nothing is displayed).

contports : is used by the `cont` command to know which ports should be displayed during a `cont` command. By default, this option contains the empty string (nothing is displayed).

spyports : is used by the `spy` command to know on which ports the breakpoints should be set. By default, this option contains the string « cerfRY » (*i.e.* all ports).

5.4.6 Commands

Let's start first by a quick overview of all commands available in Prolog IV's debugger.

First, we have the progression commands, which move from port to port: `step`, `next`, `cont`.

Then, we have the breakpoint management commands: `spy`, `unspy`.

The information display commands: `printvar`, `subdomain`, `chpt`, `where`, `up`, `down`.

and finally, various commands : `prolog`, `kill`, `help`.

We call *context* or *calling context* the state of the demonstration at a given point. A context may be seen as a selected rule, in which the state of variables depends on the calls which have already been performed in the rule's tail. Contexts are naturally linked in the structure of embedded calls (only one call is active at a given time). The commands `up` and `down` are available to surf between contexts.

step

This is the basic command to follow the step-by-step progression of the Prolog machine. Steps one step ahead in the program execution. Only stops at ports which have been indicated in the `stepports` option. Shortcut: `s`.

step *ports*

Assigns the value *ports* to the option `stepports`, and then executes a `step` command. Shortcut: `s`.

next

Goes from call to call, while displaying the ports which are indicated in the option `nextports`. Only the calls which have a lower-level than the current one are displayed. If it is not possible to go to the next call (for instance because of a failure), this command behaves like `step`. Shortcut: `n`.

next *ports*

Assigns the value *ports* to the option `nextports`, and then executes a `next` command. Shortcut: `n`.

cont

Continues the execution, while displaying the ports which are indicated in the `contports` option. Stops if a breakpoint is detected. Shortcut: `c`.

cont *ports*

Assigns the value *ports* to the option `contports`, and then executes a `cont` command. Shortcut: `c`.

where

Prints the call stack, from the most recent to the oldest. Shortcut: `w`.

source 1 *or* 0

Activates or deactivates the debugger's « *source* » mode. Only meaningful when the debugger window is used in the graphical environment. Defaults to 0 when the debugger is used without the graphical environment, and to 1 when the debugger window exists.

pbox 1 *ou* 0

Activates or deactivates the display of the debugger's status lines. When deactivated, does not show any more the textual output for the encountered ports; this is only interesting when using the graphical environment, in which case it allows the user to focus on the source code. By default, the status lines are displayed (1).

kill

Abandons the execution of the current query, and returns to the Prolog IV prompt. The `debug` mode remains active. Shortcut: `k`.

no_debug

Abandons the `debug` mode. The next command entered allows the user to abandon the currently running program (with `kill`) or to continue it (with `cont` or any other option which moves ahead).

halt

Terminates the execution of Prolog IV. The user is returned to the operating system prompt.

up

Moves the context pointer to the next context up (*i.e.* the calling rule). If already in the top context (*i.e.* the query), a warning message is displayed and no further action is taken. Shortcut: `u`.

down

Moves the context pointer to the next context down (*i.e.* the called rule). If already at the bottom context, a warning message is displayed and no further action is taken. Shortcut: `d`.

printvar

Displays all the variables in the current context. In conjunction with the `up` and `down` commands, allows a user to examine the contents of the variables which are involved in the current demonstration. Be careful, this command does not print information about subdomains (refer to the `subdomain` command for that). This command is based on the `write` primitive. Shortcut: `p`.

printvar ***var ...***

Same as `printvar`, but only displays the listed variables. Shortcut: `p`.

subdomain

Displays the subdomains of all the variables in the current context (positioned by `up` and `down`). The information is displayed in a way which is much nicer and more detailed than the output of `printvar`. Shortcut: `sd`.

subdomain ***var ...***

Same as `subdomain`, but only displays the listed variables. Shortcut: `sd`.

help

Displays the list of available debugger commands, with a short description. Shortcut: `h`.

help ***cmd ...***

Displays a short description of the commands given as arguments. Shortcut: `h`.

chpt

Displays the choice point stack, from the most recent to the oldest.

spy

Sets a breakpoint on the current rule block, for the ports specified in the `spyports` option.

spy ***name ...***

Sets a breakpoint on the rule block named *name*, for the ports specified in the `spyports` option. The name can be of form *identifier/arity* or *identifier*. In this latest case, all the blocks with name *identifier* will be spied on, regardless of their arity.

unspy

Removes a breakpoint from the current rule block.

unspy ***name ...***

Removes a breakpoint from the block named *name*.

spyports *ports*

Sets the `spyports` option used in the `spy` commands to the value *ports*. Without argument, displays the current value of the `spyports` option.

prolog

Starts a new embedded Prolog IV session. The Prolog IV prompt is displayed, and the user can type new queries. The user can then perform any desired checks or run a program, without stopping the current debugging session. This embedded session should be finished using the `stop` primitive, after which the debugger prompt is displayed.

stepports *ports*

Sets the `stepports` option used in the `step` commands to the value *ports*. Without argument, displays the current value of the `stepports` option.

nextports *ports*

Sets the `nextports` option used in the `next` commands to the value *ports*. Without argument, displays the current value of the `nextports` option.

contports *ports*

Sets the `contports` option used in the `cont` commands to the value *ports*. Without argument, displays the current value of the `contports` option.

5.4.7 Primitives and compilation

We here describe how to get in and out of the debug mode, and how to compile a program in order to make it debuggable during its execution.

Mode switching

debug, no_debug The Prolog IV command `debug` switches the Prolog IV machine into debug mode, regardless of the previous execution mode. Similarly, the `no_debug` command switches the Prolog IV machine into regular (no debug) mode, regardless of its previous state.

In order to debug a part of a program execution, one may bracket a section of code between two `debug` and `no_debug` instructions; however, no information will be available about the part of the execution which occurred in `no_debug` mode. Finally, there is no interest in using the debug mode if the program has not been compiled with the debug option on.

Compilation

In order to compile a program with the informations needed by the debugger, one needs to use:

- either the `:- debug.` directive in the file (before the first rule block),
- or the compiler option `debug(on)`.

It is possible to compile parts of a file (*i.e.* some rule blocks) in different modes by placing several `:-debug` and `:-no_debug` directives in the file.

5.4.8 Problems and limitations

- The debugger only considers the programs entered in Prolog IV through the compilation commands, with the debug optoin turned on.
- As a consequence, it is not possible to debug programs entered using the `(re)consult` or `assert` primitives.
- The meta-calls, such as those performed by `call`, « `;` » or `freeze` are ignored by the debugger.