4

Predefined predicates in Prolog IV

THE PRESENT CHAPTER includes the exhaustive list of Prolog IV's predefined predicates[1]. The list is ordered alphabetically, in order to make the search simpler.

4.1 Preliminaries

The present section gives a quick outlook on the various categories of primitives described in this chapter.

4.1.1 Rule input

Rule blocks

We call name and arity of a rule the name and arity of the head of the rule. We can then define a *rule block* as being a set of rules which have the same name and arity. Rule blocks are therefore identified by a couple $(name, arity)$. Usually, in Prolog, such rules are grouped together rather than split within a file or across a file[2]. The name of a rule block will often be denoted by $name/arity$.

Compiling and consulting

There are in Prolog IV three different ways to enter rules:

- by compiling a file,
- by consulting a file (or the console, which simply is a special file),
- by asserting rules (represented as terms) during the execution of a Prolog IV program.

There are in Prolog IV two distinct types of encoding, because of contradictory technical needs:

[1] This does not include the predefined predicates from the ISO Prolog standard, which are described in a separate chapter.

[2] There are exceptions to this rule, and the ISO standard provides some options to deal with these exceptions.

- The compiled form (or P-Code) presents the advantage of a good execution speed. Its major drawback is the impossibility to access the rules once they are compiled, to decompile them or to add rules to a block.
- The interpreted form (or asserted from) allows the user to access the rules at any time, either to decompile them or to add rules to a block. Of course, this results in a reduced execution speed, as well as an increase in memory usage.

Rule input primitives

`consult` : reads a sequence of rules on the current input stream. If a rule block is read whose name and arity correspond to an already existing block, an error occurs, signaling the redefinition attempt. See also `reconsult`.

`consult(`*`file`*`)` : works exactly as `consult` without argument, except from the fact that it reads the rules from *file* (which must be an atom containing the name of the file).

`reconsult` : reads a sequence of rules on the current input stream. It works like `consult`, except that it allows the redefinition of user-input rules. When a rule block is read whose name and arity correspond to an already existing block, no error occurs. Of course, the user is not allowed to redefine the predefined primitives and relations.

`reconsult(`*`file`*`)` : works exactly as `reconsult` without argument, except from the fact that it reads the rules from *file* (which must be an atom containing the name of the file).

`compile(`*`file`*`)` : works exactly like `consult(`*`file`*`)`, except that it encodes rules in the compiled form.

`recompile(`*`file`*`)` : works exactly like `reconsult(`*`file`*`)`, except that it encodes rules in the compiled form.

In all cases above :

- The end of the file or the term « `end_of_file.` » end the input of the rules.
- The rules are encoded in memory.
- Trying to redefine predefined primitives or constraints invariably leads to an error.

Note that it is not possible to compile rules in the console; it is necessary to use a file. However, when using Unix and a `tty` terminal (like an xterm or a cmdtool), it is possible to use the pseudo-file `/dev/tty`, like in `compile('/dev/tty')`, who will wait for rules in the terminal, until an end of file is reached (usually by typing `CTRL-D`).

4.1.2 Intervals and meta-predicates

When using domain reduction techniques to solve numerical problems, the program usually consists in two phases:

1. the constraint setup phase, which mostly uses the Prolog IV relations to add constraints,

2. the enumeration phase, whose goal is to find the solutions to the system, or to prove that there aren't any.

This enumeration phase is made necessary in almost all problems, since the informations gathered by the approximated solver usually are not sufficient.

Let's assume that we want to solve the system $x = -x$. Here is what we get when entering it as is to Prolog IV:

```
>> X = .-. X.
X ~ real.
```

This answer is just not convincing. It gets much better if we restrain the domain a bit:

```
>> X = .-. X, le(X,0).
X = 0.

>> X = .-. X, gt(X,0).
false.
```

We can note that $\{X \leq 0\}$ and $\{X > 0\}$ make a partition of the set of real numbers, which means that the solution to our constraint is the union of the solutions in these two queries, *i.e.* $\{0\}$.

This very short example shows very well the basic principle of enumeration: split a difficult problem into smaller subproblems which, hopefully, will be easier to solve. With intervals, the natural way to split problems is to split the intervals which represent the domains of the variables, like we did in the example above.

There are of course many different manners to perform an enumeration. Prolog IV includes in its primitives various tools:

1. Some are basic tools to build your own enumerations. They are low-level meta-predicates, which allow the programmer to access directly the variables' domains, to handle IEEE rationals, or to get some statistics about the interval resolution.

2. Some are pre-programmed enumerations. They are high-level meta-predicates, which allow the programmer to enumerate without having to program. Such meta-predicates are available to enumerate over Booleans, integers or reals.

The principles underlying these three predefined enumeration predicates are constant. The enumerations rely on three fundamental parameters: the stopping condition, the variable selection algorithm which chooses the next variable whose domain will be split, and the way in which this splitting is performed.

Here is the skeleton of the Prolog IV program which performs this enumeration:

```
my_enum(L):-
   stopping_condition(L),
   !.
my_enum(L):-
   select_variable(L,X),
   middle(X,M),
   split(X,M),
   my_enum(L).

split(X,M):-
   le(X,M).
split(X,M):-
   gt(X,M).
```

With :

- `stopping_condition(L)`, a predicate which succeeds when the stopping condition has been reached.
- `select_variable(L,X)`, a predicate which unifies `X` with the variable (taken from `L` which is the next to be split.
- `middle(X,M)` a predicate which computes the middle (*i.e.* splitting point) of variable `X`.

The actual implementation of Prolog IV's enumeration meta-predicates is slightly more complex, but the small program above is a good example of the methodology which is used.

4.2 Alphabetical list

boolsplit/1
boolsplit/2 — Boolean enumeration

Description: **boolsplit**(a, b) enumerates the Booleans in the domain of the elements of the list a and unifies them with the elements of a in the order of the list. The first Boolean value used is b (which may be 0 (false) or 1 (true)). If b is a free variable, it is unified with the default value, *i.e.* 0.

boolsplit(a) enumerates the Booleans in the domain of the elements of the list a and unifies them with the elements of a in the order of the list. The first Boolean value used is 0 (false).

In both cases, an error occurs when a is not a list of variables which are compatible with real numbers.

Examples:

```
>> or(B,C), boolsplit([B,C]).
C = 1,
B = 0;
C = 0,
B = 1;
C = 1,
B = 1.
>> or(B,C), boolsplit([B,C],1).
C = 1,
B = 1;
C = 0,
B = 1;
C = 1,
B = 0.
>> A ~ band(B,C), boolsplit([A,B,C]).
C = 0,
B = 0,
A = 0;
C = 1,
B = 0,
A = 0;
C = 0,
B = 1,
A = 0;
C = 1,
B = 1,
A = 1.
```

See also: intsplit/3, realsplit/4.

bounds/3 — Bounds of a domain

Description: **bounds**(a, b, c) is used to get the bounds of a's domain. The lower and upper bounds of the domain are unified respectively with b and c.

A call to **bounds**(a, b, c) fails in the following cases:

- a's domain is not compatible with a numerical domain,
- the domain of a is unbounded on at least one side.

By combining these conditions, we get that a must be a bounded numerical variable.

It is important to note that the values returned by **bounds/3** are lower and upper bounds. This means that these values do not necessarily belong to the domain of the variable, in particular when the variable is constrained to belong to an open interval.

Examples:

```
>> bounds(1, B, C).
C = 1,
B = 1.
>> bounds(cc(1,2), B, C).
C = 2,
B = 1.
>> bounds(oo(1,2), B, C).
C = 2,
B = 1.
>> bounds(pi, B, C).
C = 13176795/4194304,
B = 6588397/2097152.
>> bounds(le(0), B, C).
false.
>> bounds(ge(0), B, C).
false.
```

See also: glb/2, lub/2.

compile/1
compile/2 — Compilation of a file

Types: compile(+atom)
compile(+atom, +list)

Description: **compile**(a) reads a sequence of rules in the file named a (a is an atom whose label is the name of the file). The rules are compiled and stored in memory. If a rule block with same name and arity already exists in memory, an error occurs indicating the redefinition attempt. To solve this problem, refer to **recompile/1**.

In the two-argument version **compile**(a, b), the argument b is a list of options. This list may contain the following options:

- `debug(off)`, `debug(on)` which are used in order to control the generation of debugging information in the compiled program. When the flag is `on`, the information is generated, and many program optimisations, such as indexing, are lost. This results in a reduced performance. By default, the debug flag is `off`.
- `syntax(prolog4)`, `syntax(iso)` which are used to control the syntactical mode in which the rules are read. By default, the current syntactical mode is used (*i.e.* the one which is indicated by the prompt).

These options therefore supercede the default mode. There is a way to change their value within a file by using compiler directives in the file. For instance, the directive `:- syntax(prolog4).` can be used to switch to the `prolog4` syntactical mode within a file (regardless of the current mode or of the flag values); the rules which follow the directive will be analysed using the `prolog4` mode.

When an error occurs during the compilation, the rule base remains unchanged (*i.e.* as it was before the call to `compile`.

Examples:

```
>> compile('menu.p4').
true.
>> compile('menu.p4', [debug(on)]).
### error (procedure lightMeal/3) : illegal redefinition
    of a static procedure.
error: normal_error
>> recompile('menu.p4', [debug(on)]).
true.
```

See also: recompile/n, consult/n, debug/0.

consult/0
consult/1 — Consultation of a file

Types: consult
consult(+atom)

Description: **consult**(a) reads a sequence of rules in the file named a (a is an atom whose label is the name of the file). The rules are analysed and stored in memory. If a rule block with same name and arity already exists in memory, an error occurs indicating the redefinition attempt. To solve this problem, refer to **reconsult/1**.

The rules input with `consult` are *dynamic*, which means that the rule management primitives (like `retract`, `assert` or `clause`) may be applied to them.

The rules are not compiled (this is the major difference with `compile`. The rules are simply added to the rule base using the `assert` primitive. This has several drawbacks: the execution of such rules is quite slow, and the debugger cannot be used to trace their execution.

In the version without argument, the default file is the current input stream (by default, the console).

In case of error during the consultation, all the valid rules entered before the occurrence of the error are added to the rule base. Of course, the rule with the error is eliminated.

Examples:
```
>> consult('menu.p4').
true.
>> consult('menu.p4').
Consulting ...
error: consult_error(error(permission_error(modify,
static_procedure,lightMeal(_857,_858,_859)),assert/2))
>> reconsult('menu.p4').
true.
```

See also: reconsult/n, compile/n.

cpu_time/1 — CPU time measurement

Types: cpu_time(?T)

Description: **cpu_time(T)** unifies T with the CPU time (in milliseconds) spent since the last call to `reset_cpu_time`.

Examples:
```
>> reset_cpu_time, work, cpu_time(T).
T = 16500. % soit 16,5 seconde CPU
>>
```

See also: reset_cpu_time/0.

debug/0 — Switching to debug mode

Types: debug

Description: **debug** makes the debugger active. After a call to **debug**, the debugger starts collecting information, and watching the execution of programs (*e.g.* looking for breakpoints). This therefore leads to a loss of performance. Of course, the information can be fully collected only with programs which have been compiled with the debug flag on (which excludes the consulted programs and the programs which have been copmiled with the debug flag off).

Please refer to the « Environment » chapter for further information about the debugger.

See also: no_debug/0, compile/n.

def_array/2 ——— Array definition

Types: def_array(+atom, +integer)

Description: **def_array**(a, b) dynamically defines a table of Prolog constants named a and containing b elements. This table will act as « global » (which may be accessed from anywhere) and « static » (which resists backtracking). The possible values of the array index are $\{1, \ldots, b\}$.

If an array with the same name already exists, then:

- if the arrays have the same size, then nothing happens;
- if the sizes are different, then an error occurs.

The access to the array elements, and the assignment of values to these elements are similar to what happens in other programming languages.

Examples:

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10;
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

See also: record/2, recorded/2, undef_array/1.

ebounds/3 ——— Bounds of a domain

Description: **ebounds**(a, b, c) is used to compute the lower and upper bounds of a's domain, in the extension of reals. The lower and upper bounds are unified respectively with b and c.

A call to **ebounds**(a, b, c) fails if a's domain is not compatible with a numerical type.

An extended real can take the following values:

- A real number x,
- $+\infty$, which is denoted by `pinfinity` in Prolog IV,
- $-\infty$, which is denoted by `minfinity` in Prolog IV,

- for every real number x, the object x^+, which is strictly greater than x, and strictly lower than any real numbers strictly greater than x. We denote x^+ by up(x) in Prolog IV.
- for every real number x, the object x^-, which is strictly lower than x, and strictly greater than any real numbers strictly lower than x. We denote x^- by dn(x) in Prolog IV.

From a more practical point of view, the extended reals are used in two cases:

- in order to make a bounds analysis which works even if a variable is unbounded;
- in order to know if an interval is open or closed.

Even more practically, we can build a table which shows all possible types of intervals, and see the extended bounds for all of them:

Interval	*Lower bound*	*Upper bound*
$[a, b]$	a	b
$[a, b)$	a	dn(b)
$(a, b]$	up(a)	b
(a, b)	up(a)	dn(b)
$[a, +\infty)$	a	pinfinity
$(a, +\infty)$	up(a)	pinfinity
$(-\infty, b]$	minfinity	b
$(-\infty, b)$	minfinity	dn(b)
$(-\infty, +\infty)$	minfinity	pinfinity

This table may be summarised in the following four statements :

- ebounds/4 returns `minfinity` when a variable has no lower bound,
- ebounds/4 returns `pinfinity` when a variable has no upper bound,
- ebounds/4 returns up(a) when a variable has an unreachable lower bound,
- ebounds/4 returns dn(a) when a variable has an unreachable upper bound.

Examples:

```
>> ebounds(1,B,C).
C = 1,
B = 1.
>> ebounds(cc(1,2),B,C).
C = 2,
B = 1.
>> ebounds(oo(1,2), B, C).
C = dn(2),
B = up(1).
>> ebounds(pi, B, C).
C = dn(13176795/4194304),
B = up(6588397/2097152).
>> ebounds(ge(0), B, C).
C = pinfinity,
B = 0.
>> ebounds(ge(0), B, C).
C = pinfinity,
B = 0.
```

See also: eglb/2, elub/2.

eglb/2 — Lower bound

Description: **eglb**(a, b) is used to compute the greatest lower bound of a, as an extended real. The result is unified with b.

A call to **eglb**(a, b, c) fails if a's domain is not compatible with a numerical type.

Please refer to **ebounds/3** for a detailed description and important remarks concerning extended reals.

Examples:

```
>> eglb(1, B).
B = 1.
>> eglb(cc(1,2), B).
B = 1.
>> eglb(oo(1,2), B).
B = up(1).
>> eglb(pi, B).
B = up(6588397/2097152).
>> eglb(ge(0), B).
B = 0.
>> eglb(le(0), B).
B = minfinity.
```

See also: ebounds/3, elub/2.

elub/2 — Upper bound

Description: **elub**(a, b) is used to compute the least upper bound of a, as an extended real. The result is unified with b.

A call to **elub**(a, b, c) fails if a's domain is not compatible with a numerical type.

Please refer to **ebounds/3** for a detailed description and important remarks concerning extended reals.

Examples:

```
>> elub(1, B).
B = 1.
>> elub(cc(1,2), B).
B = 2.
>> elub(oo(1,2), B).
B = dn(2).
>> elub(pi, B).
B = dn(13176795/4194304).
>> elub(ge(0), B).
B = pinfinity.
>> elub(le(0), B).
B = 0.
```

See also: ebounds/3, eglb/2.

enum/1
enum/3 — Integer enumeration

Description: **enum**(a, b, c) enumerates all the integers of a's domain which are between b and c, and unifies them with a. The order in which the enumeration is performed is not guaranteed.

enum(a) enumerates all the integers of a's domain, and unifies them with a. The order in which the enumeration is performed is not guaranteed.

It is possible to define **enum/3** from **enum/1** as follows:

```
enum(X,L,U) :-
  cc(X, L, U),
  enum(X).
```

For both predicates, an error occurs if any of their arguments is not compatible with real numbers.

Examples:

```
>> X ~ cc(0,5), enum(X).
X = 0;
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
>> enum(X, .-. pi, pi).
X = -3;
X = -2;
X = -1;
X = 0;
X = 1;
X = 2;
X = 3.
>> enum(toto).
error: error('Real variable expected',enum)
```

See also: enumlin/3, intsplit/3.

enumlin/1
enumlin/3 — Integer enumeration

Description: **enumlin**(a, b, c) enumerates all the integers of a's domain which are between b and c, and unifies them with a. The order in which the enumeration is performed is not guaranteed.

enumlin(a) enumerates all the integers of a's domain, and unifies them with a. The order in which the enumeration is performed is not guaranteed.

It is possible to define **enumlin/3** from **enumlin/1** as follows:

```
enumlin(X,L,U) :-
  gelin(X, L),
  lelin(X, U),
  enumlin(X).
```

For both predicates, an error occurs if any of their arguments is not compatible with real numbers.

Examples:

```
>> X ~ gelin(0) n lelin(5), enumlin(X).
X = 0;
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
>> enumlin(X, -3, 3).
X = -3;
X = -2;
X = -1;
X = 0;
X = 1;
X = 2;
X = 3.
>> enumlin(toto).
error: error('Real variable expected',enumlin)
```

See also: enumlin/3, intsplit/3.

float_rank/2 — Rank of a floating-point number

Description: **float_rank**(a, b) establishes a relation between a floating-point number a and its rank b (see definition below). When calling this predicate, at least one of the arguments must be bound to a value. Furthermore, if a has a value which is not an IEEE single-precision floating-point number, or if b has a value which does not represent the rank of a floating-point number, a typing error occurs.

The rank of a floating-point number is a numbering of these numbers, such that the rank of 0 is 0. The smallest positive floating-point number has rank 1, and the largest negative one (*i.e.* the closest to 0) has rank -1.

This numbering may be used in order to determine the number of floating-point numbers between two numbers.

Examples:

```
>> float_rank(0, B).
B = 0.
>> float_rank(B, 1), min_positive_float(B).
B = 1/713623846352979940529142984724747568191373312.
>> float_rank(1,X).
X = 1065353216.
>> float_rank(1,X1), float_rank(2, X2), Diff = X2 - X1 .
Diff = 8388608,
X2 = 1073741824,
X1 = 1065353216.
>> float_rank(A, B).
error: error(instantiation_error,float_rank/2)
>> float_rank(1/3, B).
error: error(type_error('ieee754(single)',1/3),float_rank/2)
```

See also: max_float/1, min_positive_float/1.

float_size/2 — Measuring the floats in a domain

Description: **float_size**(a, b) returns in b a measurement of a's domain, which is the number of atomic IEEE intervals (*i.e.* the intervals whose bounds are two identical or successive IEEE numbers).

float_size/2 is used to measure the size of a variable's domain, which is useful in particular during the selection of a variable on which enumeration will occur.

In simple intervals mode, if L is the rank of a's lower bound, U the rank of a's upper bound, and N the number of these bounds that a cannot reach, then b may be computed by the formula:

$$b = (U - L) \times 2 - N + 1$$

In the unions of intervals mode, the measure of a variable's domain is th sum of the measures of the intervals which compose it.

Examples:

```
>> float_size(1,S).
S = 1.
>> float_size(pi,S).
S = 1.
>> float_size(real,S).
S = 8556380159.
>> float_size(oo(1,2), S).
S = 16777215.
>> float_size(cc(1,2), S).
S = 16777217.
>> cc(X,1,2), float_size(X,S),
     float_rank(1,S1), float_rank(2,S2),
     S = (S2 - S1)*2 + 1 .
S2 = 1073741824,
S1 = 1065353216,
S = 16777217,
X ~ cc(1,2).
>> set_prolog_flag(interval_mode,union).
true.
>> float_size( pi u 2 .*. pi, S).
S = 2.
>> float_size(cc(1,1000) n int, S).
S = 1000.
>> float_size(cc(1,2) u cc(4,7), S).
S = 29360130.
>> set_prolog_flag(interval_mode,simple).
true.
```

See also: int_size/2, real_size/2, float_rank/2.

freeze/2 — Delaying the execution of a goal

Types: freeze(?term, +goal)

Description: **freeze**(a, b) delays the execution of goal b until the value of a's label is not known. By « known », we mean the fact that this label can only take one value. This notion is the one which is used by **nonvar/1**.

Note: When a becomes known, the set of all goals frozen on a is inserted at the beginning of the list of goals to be executed. For efficiency reasons, the order in which these goals are executed is not specified.

Examples:

```
>> freeze(X, write(toto)).
X ~ tree.
>> freeze(X, write(toto)), X = 1 .
totoX = 1.
```

A delayed goal is not unfrozen when a variable is constrained but not enough to bind its label:

```
>> freeze(X, dif(X,1)), X ~ cc(1,3).
X ~ cc(2,3).
```

On the other hand, the goal is unfrozen as soon as this value is known:

```
>> freeze(X, dif(X,1)), X = 1 .
false.
```

The value does not need, however, to be completely known (the label is sufficient). Here is an example of this, in which we can also note the use of parentheses in order to put a sequence of goals as argument to **freeze/2**:

```
>> freeze(X, (write(toto),nl)), X = toto(Y).
toto
X = toto(Y),
Y ~ tree.
```

See also: nonvar/1.

get_istats/3
get_istats/2 — Getting interval statistics

Description: **get_istats**(a, b) unifies a and b respectively with the value of the following counters:

- The number of calls to the fixpoint algorithm,
- The number of primary relation evaluations performed in the fixpoint algorithm.

get_istats(a, b, c) unifies a, b and c respectively with the value of the following counters:

- The number of calls to the fixpoint algorithm,
- The number of primary relation evaluations performed in the fixpoint algorithm.
- The size of the largest union of intervals.

Examples:

```
>> reset_istats, intsplit([cc(1,4), cc(3, 6)]), false.
false.
>> get_istats(A, B).
B = 253,
A = 93.
>> get_istats(A, B, C).
C = 0,
B = 253,
A = 93.
>> set_prolog_flag(interval_mode,union).
true.
>> X ~ square(int n cc(1,10)).
X ~ 1 u 4 u 9 u 16 u 25 u 36 u 49 u 64 u 81 u 100.
>> get_istats(A,B,C).
C = 10,
B = 268,
A = 99.
>> set_prolog_flag(interval_mode,simple).
true.
```

See also: get_istats/2, reset_istats/0.

glb/2 — Lower bound

Description: **glb**(a, b) is used to get the greatest lower bound of a's domain; this value is then unified with b.

A call to **glb**(a, b) fails in the following cases:

- a's domain is not compatible with a numerical domain,
- the domain of a does not have a lower bound.

By combining these conditions, we get that a must be a numerical variable with a lower bound.

It is important to note that the value returned by **glb/2** is a lower bound. This means that this value does not necessarily belong to the domain of the variable, in particular when the variable is constrained to belong to an open interval.

Examples:

```
>> glb(1,B).
B = 1.
>> glb(cc(1,2), B).
B = 1.
>> glb(oo(1,2), B).
B = 1.
>> glb(pi, B).
B = 6588397/2097152.
>> glb(le(0), B).
false.
>> glb(ge(0), B).
B = 0.
```

See also: bounds/3, lub/2.

int_size/2 — Integer domain measurement

Description: **int_size**(a, b) unifies b with a measurement of a's domain, which is the number of integer numbers included in a. If the domain is unbounded, then b is unified with `pinfinity`.

int_size/2 may be used to measure the size of a variable's domain, and it is particularly useful in the variable selection phase of an enumeration.

Examples:

```
>> int_size(1,S).
S = 1.
>> int_size(pi,S).
S = 0.
>> int_size(real,S).
S = pinfinity.
>> int_size(cc(1,2), S).
S = 2.
>> int_size(oo(1,2), S).
S = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> int_size(cc(1,1000) n int, S).
S = 1000.
>> int_size(cc(1,2) u cc(4,7), S).
S = 6.
>> set_prolog_flag(interval_mode,simple).
true.
```

See also: float_size/2, real_size/2.

intsplit/1
intsplit/2
intsplit/3 — Integer enumeration

Description: **intsplit**(a, b, c) enumerates the integers in the domain of the elements of list a. The b and c arguments are control parameters; b is used at each step to select the next variable whose domain will be split:

- $b =$ `smallest_domain` : Selection of the variable with the smallest domain,
- $b =$ `greatest_domain` : Selection of the variable with the largest domain,
- $b =$ `list_order` : Selection of the first variable of the list which does not verify the stopping condition (see below),
- $b =$ *my_choice* : The selection is made by the predicate *my_choice*(a, x), which needs to be defined by the user; a is the original list, and the predicate must unify with x one of a's variables.

c's value is used to determine the stopping condition :

- $c = \texttt{depth}(n)$: The enumeration stops when the search tree's depth gets up to n, *i.e.* after making n choices.
- $c = \texttt{prec}(p)$, or $c = p$: Stops when all the variables of a have an integer domain measure which is smaller than p, *i.e.* when their domain contains at least p integers.
- $c = \mathit{my\_stop}$: Uses the user-defined predicate *my_stop*(a) to decide whether or not to stop; a is the original variable list, and the predicate should succeed only when the stopping condition is reached.

When user-defined predicates are used, the list passed as arguments (for both variable selection and stop tests) contains the current state of the list originally passed as argument to **intsplit**. The user may perform any operations on this list, including any operations on the variable. No check whatsoever is performed by Prolog IV, so this option should be reserved to advanced users, because an inappropriate operation may lead to infinite loops or other problems.

intsplit(a, b) enumerates the integers in the domain of the elements of the list a. The argument b is used to control the selection of the next variable to split, like for **intsplit/3**. The algorithm stops when the domain of each variable is reduced to a single integer. **intsplit/2** may be written using **intsplit/3** as follows:

```
intsplit(A, B) :-
  intsplit(A, B, prec(1)).
```

intsplit(a) enumerates the integers in the domain of the elements of the list a. At each step, the variable with the smallest domain is selected for splitting. The algorithm stops when the domain of each variable is reduced to a single integer. **intsplit/1** may be written using **intsplit/3** as follows:

```
intsplit(A) :-
  intsplit(A, smallest_domain, prec(1)).
```

Examples:

```
>> A ~ cc(1,20), intsplit([A], smallest_domain, prec(5)).
A ~ cc(1,5);
A ~ cc(6,10);
A ~ cc(11,15);
A ~ cc(16,20).
>> A ~ cc(1,20), B ~ cc(1,20), B ~ square(A),
    intsplit([A,B]).
B = 1,
A = 1;
B = 4,
A = 2;
B = 9,
A = 3;
B = 16,
A = 4.
>> A ~ cc(1,20), B ~ cc(1,20), B ~ square(A),
    intsplit([A,B], greatest_domain, depth(1)).
B ~ cc(1,4),
A ~ cc(1,2);
B ~ cc(9,16),
A ~ cc(3,4).
```

See also: boolsplit/2, realsplit/4.

lub/2 — Lest upper bound

Description: **lub**(a, b) is used to get the least upper bound of a's domain; this value is then unified with b.

A call to **lub**(a, b) may fail in the following cases:

- a has a type which is not compatible with real numbers,
- a's domain does not have a least upper bound.

The two conditions above mean that a must be a numerical variable with an upper bound.

One must note that the values returned by **lub/3** are upper bounds. These bounds may not always be inside the domain of the variable, in particular in the case of open intervals.

Examples:

```
>> lub(1, B).
B = 1.
>> lub(cc(1,2), B).
B = 2.
>> lub(oo(1,2), B).
B = 2.
>> lub(pi, B).
B = 13176795/4194304.
>> lub(le(0), B).
B = 0.
>> lub(ge(0), B).
false.
```

See also: bounds/3, glb/2.

max_float/1 — Largest IEEE rational

Description: **max_float**(a) unifies a with the largest IEEE rational, *i.e.* 340282346638528859811704183484516925440.

Examples:

```
>> max_float(A).
A = 340282346638528859811704183484516925440.
```

See also: float_rank/2, min_positive_float/1.

min_positive_float/1 — Smallest positive IEEE rational

Description: **min_float**(a) unifies a with the smallest positive IEEE rational, *i.e.* $\frac{1}{713623846352979940529142984724747568191373312}$.

Examples:

```
>> min_positive_float(A).
A = 1/713623846352979940529142984724747568191373312.
```

See also: float_rank/2, max_float/1.

no_debug/0 — Ending debug mode

Types: no_debug

Description: **no_debug** makes the debugger inactive. All the debugger's functions are deactivated (watching breakpoints, displaying calls, ...).

Please refer to the « Environment » chapter for further details about debugging.

See also: debug/0, compile/n.

real_size/2 — Real domain measurement

Description: **real_size**(a, b) unifies b with a measure of a's domain, which is the size of a's domain in **R**. If this domain is not bounded, then b is unified with `pinfinity`.

real_size/2 is used to measure the size of a variable's domain, in particular during the variable selection phase in an enumeration.

In simple intervals mode, if L is the value of a's domain's greatest lower bound, and U is the value of a's domain's least upper bound, then the measure b may be computed by:

- $b = U - L$ if U and L both exist,
- `pinfinity` otherwise.

In unions of intervals mode, the measure of a variable's domain is the sum of the measures of the intervals in the union.

Examples:

```
>> real_size(1,S).
S = 0.
>> real_size(pi,S).
S = 1/4194304.
>> real_size(real,S).
S = pinfinity.
>> real_size(cc(1,2),S).
S = 1.
>> real_size(oo(1,2),S).
S = 1.
>> set_prolog_flag(interval_mode,union).
true.
```

In the example below, we can see that the intervals which are reduced to a singleton have a 0 size (even if there are 1000 of them).

```
>> real_size(cc(1,1000) n int, S).
S = 0.
>> real_size(cc(1,2) u cc(4,7), S).
S = 4.
>> set_prolog_flag(interval_mode,simple).
true.
```

See also: float_size/2, int_size/2.

realsplit/1
realsplit/2
realsplit/3
realsplit/4 — Real enumeration

Description: **realsplit**(a, b, c, d) enumerates the real numbers in the domain of the elements of list a. The domain of each variable is split into two parts at each iteration. Of course, the resolution of this enumeration cannot exceed the precision of the IEEE rationals. A variable is selected in the list a for splitting at each step, according to b's value:

- $b =$ `smallest_domain` : Selection of the variable with the smallest domain,
- $b =$ `greatest_domain` : Selection of the variable with the largest domain,
- $b =$ `list_order` : Selection of the first variable of the list which does not verify the stopping condition (see below),
- b = *my_choice* : The selection is made by the predicate *my_choice*(a, x), which needs to be defined by the user; a is the original list, and the predicate must unify with x one of a's variables.

c's value is used to determine the stopping condition :

- $c =$ `depth(`n`)` : The enumeration stops when the search tree's depth gets up to n, *i.e.* after making n choices.

- c = `prec(`p`)`, or $c = p$: Stops when all the variables of a have a domain measure which is smaller than p. The measurement method is defined by the argument d.
- c = *my_stop* : Uses the user-defined predicate *my_stop*(a) to decide whether or not to stop; a is the original variable list, and the predicate should succeed only when the stopping condition is reached.

d's value is used to determine the domain measurement method:

- d = `real_mode` : Measurement over **R** (refer to **real_size/2**),
- d = `float_mode` : Measurement over floating-point numbers (see **float_size/2**).

The measurement method is used both for selecting the variable on which the next splitting will be performed and for making the stopping tests.

If some of the arguments b, c and d are not instantiated when calling **realsplit/4**, then they are unified with the selected default value. This feature may be used in order to forbid one method (by using **dif/2**).

realsplit(a, b, c) works like its four-argument counterpart, except that the measurement method always is the real measurement (see **real_size/2**), which corresponds to the `real_mode` from **realsplit/4**.

realsplit(a, b) worls like its three-argument counterpart, except that the enumeration stops when the precision of the computation reaches 10^{-4}.

realsplit(a) works like its two-argument counterpart, except that the variable selected for splitting always is the one with the smallest domain, which corresponds to the `smallest_domain` option.

These three versions can easily be defined from **realsplit/4** as follows:

```
realsplit(A, B, C) :-
  realsplit(A, B, C, real_mode).

realsplit(A, B) :-
  realsplit(A, B, prec(1/10000), real_mode).

realsplit(A) :-
  realsplit(A, smallest_domain, prec(1/10000), real_mode).
```

Examples:

```
>> A ~ cc(1,pi), realsplit([A], B, prec(1)).
B = smallest_domain,
A ~ cc(1,'>1.5353982');
B = smallest_domain,
A ~ oc('>1.5353982','>2.0707964');
B = smallest_domain,
A ~ oo('>2.0707964','>2.6061944');
B = smallest_domain,
A ~ co('>2.6061944','>3.1415927').
```

See also: boolsplit/2, intsplit/3.

recompile/1
recompile/2 — Recompiling a file

Types: recompile(+atom)
recompile(+atom, +list)

Description: **recompile/1** is very similar to the **compile/1** primitive. **recompile**(a) reads a sequence of rules in a file named a. The rules are compiled and stored in memory. If a rule block with the same name and arity already exists in the rule base, then it is simply redefined, without generating an error.

recompile/2 corresponds similarly to **compile/2**. Please refer to **compile/2** for a description of the options and examples.

See also: compile/n, consult/n, debug/0.

reconsult/0
reconsult/1 — Reconsulting a file

Types: reconsult
reconsult(+atom)

Description: **reconsult**(a) is very similar to the **consult/1** primitive. **reconsult**(a) reads a sequence of rules in q file named a. The rules are interpreted and stored in memory. If a rule block with the same name and arity already exists in the rule base, then it is simply redefined, without generating an error.

reconsult/0 corresponds similarly to **consult/0**. Please refer to **consult/0** for a full description and examples.

See also: consult/n, compile/n, recompile/n.

record/2 — Global variable assignment

Types: record(+atom, @term)

Description: **record**(a, b) associates the term b to the atom given in a. This acts just like if a was a global variable. This assignment is in fact similar to the assignment from Fortran or Pascal.

Examples:

```
>> record(age, 21).
true.
>> write(age), nl.
age
true.
>> recorded(age, X).
X = 21.
```

See also: recorded/2.

record/2 — Assignment of an array element

Types: record(+array element, @term)

Description: **record**$(a(i), b)$ associates the term b to the i-th element of array a.

Examples:

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10;
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

See also: def_array/2, recorded/2, undef_array/1.

recorded/2 — Global variable evaluation

Types: recorded(+term, ?term)

Description: **recorded**(a, b) puts in b a term which has been previously associated to a, which must be an atom or an array element. The value returned follows the rules:

- If a value has previously been assigned to the atom or array element, then this value is returned.
- Otherwise, the atom or array element itself is returned.

Examples:

```
>> record(age, 21).
true.
>> write(age), nl.
age
true.
>> recorded(age, X).
X = 21.
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10;
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

See also: def_array/2, record/2, undef_array/1.

reset_istats/0 — Resetting interval statistics

Description: **reset_istats** resets all three interval counters to zero:

- The number of calls to the fixpoint algorithm,
- The number of primary relations reevaluated in the fixpoint algorithm,
- The size of the largest union of intervals.

This primitive is to be used together with **get_istats/2** and **get_istats/3**.

Examples:

```
>> reset_istats.
true.
```

See also: get_istats/2, get_istats/3.

reset_cpu_time/0 — Resetting the clock

Types: reset_cpu_time

Description: **reset_cpu_time** resets the CPU time counter to zero. This primitive is to be used together with **cpu_time/1**.

Examples:

```
>> reset_cpu_time, cpu_time(T).
T = 0.
```

See also: cpu_time/1.

undef_array/1 — Array destruction

Types: undef_array(+atom)

Description: **undef_array**(a) destroys an array which has been previously created by **def_array**. The array a cannot be used any more without another definition through **def_array**.

Examples:

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10;
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

See also: def_array/2, record/2, recorded/2.