IF/Prolog V5.3

Constraints Package

Siemens AG Austria

Is there

anything you would like to tell us about this manual?
Please send us your comments.

Siemens AG Austria
PSE KB B3
Gudrunstrasse 11
A-1100 Vienna
Austria

Fax.: +43-1-1707 56992

email: prolog@siemens.at

Contents

Preface

The IF/Prolog system from Siemens AG Austria is an implementation of the ISO Prolog standard (ISO = International Standardization Organization). This standard was prepared by ISO Working Group 17, comprising representatives from various national standardization bodies.

IF/Prolog also contains interfaces and predicates which extend the language and ensure compatibility with earlier versions of the product.

The Standard dictates us to supply a strictly conforming mode, where the Prolog system only accepts and supplies conforming language predciates. To invoke this mode, see the section on invoking IF/Prolog in the User's Guide.

The constraints package is entirely an extension, and is not part of the ISO Prolog Standard [11].

Target group

The *IF/Prolog Constraints Package* manual is intended for anyone wishing to use **constraints** to enhance the efficiency and expressiveness of Prolog programs.

Manuals

The documentation for IF/Prolog comprises of the following manuals:

- *IF/Prolog Reference Manual*
- *IF/Prolog User's Guide*
- *IF/Prolog Windows Interfaces*
- *IF/Prolog OSF/Motif Interface*
- *IF/Prolog Constraints Package*
- *IF/Prolog Java Interface*

The *IF/Prolog Reference Manual* contains a description of the semantics, built-in predicates, C interface functions, debugger commands and environment control of IF/Prolog. It also contains an overview of the syntax of the Prolog language.

Predicates associated with an interface or package are described in the respective manuals. The *IF/Prolog User's Guide* describes how to work with Prolog on a computer running under the UNIX, DOS, Windows and other operating system.

As the set of manuals for IF/Prolog are not tutorials, you should be familiar with the Prolog language. You should also be familiar with the basics of the operating system and know how to use one of the editors installed on your computer.

Summary of contents

The *IF/Prolog Constraints Package* manual provides a short introduction describing the general characteristics of constraints and the extension of the number domain by big integers and rational numbers. In the following chapters, the individual classes of constraints for IF/Prolog are then described. You will find the built-in predicates for each constraint class described in alphabetical order in the latter part of the respective chapters.

Each chapter is divided into a descriptive and a reference part in accordance with the organization of IF/Prolog. The built-in predicates for each constraint class are defined in separate modules and consequently, after importing a given constraint module, you can use the associated predicates and operators.

Notational conventions

The following notational conventions are used throughout this manual:

$\boxed{\text{xxx}}$	Syntax definitions are enclosed within a frame.
`bind`	Prolog language elements, operating system commands and outputs from the system are printed in `teletype` font.
Name	*Italics* are used to represent variable parts in inputs and outputs where you should substitute them with your own values.
[ ]	Square brackets denote optional entries in the syntax notation; the brackets are not part of the Prolog text.
[ ]	Square brackets in bold type are elements of the Prolog list notation and are part of the Prolog text.
{ }	Braces denote alternatives in the syntax notation; the brackets are not part of the Prolog text.
|	A bar denotes alternatives in the syntax notation.
( )	Parentheses are required parts of the Prolog predicate notation and is part of the Prolog text.
...	Ellipsis indicate that the preceding syntax element may be repeated.
`atom/1`	Predicates are specified in the form *Name*/*Arity*.
`nl/0/1`	Several predicates with the same name and different arities are denoted in this form.
[1]	A number in square brackets indicates a reference to another manual or a textbook. The number identifies the publication in the Bibliography at the end of the manual.

The following pictograms are also used:

$\boxed{\textbf{i}}$	for important advice and related information.
$\boxed{\textbf{!}}$	for warnings.

Representation of built-in predicates

The descriptions of the built-in predicates have a standard format. They contain, if required, in the following order:

- Descriptive title (meaning of the predicate)
  The function of the predicate is described here in keywords.
- Predicate head (boxed)
  Contains the functor of the predicate, followed by the arguments and their call modi. Metacharacters which are explained below are used to represent the syntax.

- Full description
  Begins with a passage of text which gives a detailed explanation of how the predicate works, and the function of its arguments. If two or more similar predicates are described together, their differences will be pointed out here.

- Description of arguments
  Shows for each argument the modes of terms that are either required for instantiation (call modes `+` and `@`) or permissible (call mode `?`) when the predicate is called.
  If only a specific set of values are permitted as arguments, these are shown in the form of a list, separated by '`|`' symbols.

- Description of exceptions
  Possible runtime exceptions are described here.

- Hints
  Contains explanatory details on the use and special features of the predicate.

- Example
  Illustrates how to use the predicate.

- Reference to related predicates
  Lists cross-references to other IF/Prolog predicates.

Some of these subsections may be omitted if they are inapplicable for the predicate concerned; for example, the subsection headed '*Hints*' does not always appear.

Predicates which permit backtracking are indicated by a hash character ('`#`') in front of the functor. This character does not appear with predicates not having this capability.

For **metapredicates**, the argument list is followed by the text `[` @ +*Module* `]`. The module qualification can be specified by `@/2`. **Metapredicates** are supplied with information on the calling module.

Directives are special syntactic structures which can be specified in IF/Prolog texts and which are processed when these texts are read in (e.g. with `consult/1`). They are indicated by `:-` in front of the functor, as they would be written in a Prolog text.

User definable predicates are automatically called at particular points by IF/Prolog if the user has defined them. They are identified by the text `[ :-` *Body* `]` after the argument list.

The **call mode** specifies the instantiation of an argument at the time of the call. In front of each argument there is a sign ('`@`', '`+`', '`-`', or '`?`') to indicate the call type of the argument, as follows:

@ The argument is a pure **input** parameter. The current parameter specified in the call must be of the prescribed type and any uninstantiated variables contained in this parameter are not instantiated in the call.

`+` The argument is an **input** parameter. The current parameter specified in the call must be of the prescribed type. Any uninstantiated variables contained in this parameter may be instantiated in the call.

When the argument is an atomic term, there is no difference between the modes `+` and `@`. The mode `@` is therefore used only when the argument may be a compound term.

`?` The argument is an **input/output** parameter. The current parameter must be either a variable or a term of the prescribed type. In the course of the execution of the predicate, this parameter is unified. If this unification is not successful, then the entire predicate call will fail. Any uninstantiated variables contained in this parameter may be instantiated in the call.

`-` The argument is a pure **output** parameter. The current parameter must be an uninstantiated variable. If the predicate succeeds, this variable is instantiated with the result of the predicate call. The type of result from the predicate call is indicated in the section on '*Arguments*' in the full description.

Several call patterns are possible for some predicates.

Chapter 1

Constraints

Prolog is designed as a declarative programming language with a number of procedural language constructs. For reasons of efficiency however, many problems have been solved by means of procedural algorithms. Constraints constitute a powerful extension of the descriptive Prolog language. Constraints allow you to describe more simply dependencies within a system. For theoretical background of constraints, please refer to *Hentenryck* [15].

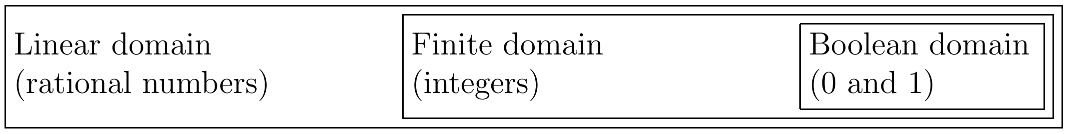
In standard Prolog, an uninstantiated variable may assume any value. The variable must be instantiated before testing can take place to determine whether consistency requirements are satisfied. Consistency conditions are thus used passively, i.e. not until the variable has a value.

You can specify constraints, however, before the Prolog system attempts any instantiations. The variables are instantiated during normal goal proving (or through constraint propagation). This may influence other variables if you have again defined constraints between these variables (**propagation**). The Prolog system automatically takes account of the newly acquired information. If only one value is left in the domain, the variable can be instantiated.

Constraints are thus conditions relating to one or more variables in a particular domain. Three domains are differentiated:

- The linear domain is infinite and contains rational numbers.
- The finite domain includes integers and is defined either as an interval, as an enumeration, or as a sequence of values.
- The Boolean domain is a subset of the finite domain and contains only the values 0 for `false` and 1 for `true`.

The domains are nested, as illustrated by the following diagram. The Boolean domain is a subset of the finite domain which is in turn a subset of the linear domain.



By analogy with this division into three domains, the constraints are also divided into different classes:

- the class of constraints for the linear domain: linear constraints
- the class of constraints for the finite domain: finite constraints
- the class of constraints for the Boolean domain: Boolean constraints

Coroutines constitute a separate class of constraints which cannot be defined by way of a separate domain. A coroutine allows you to delay the proof of a goal until a variable has been instantiated, and thus to define more complex constraints. Coroutines are, so to speak, the basis on which constraints can be developed.

A constraint is entered by using built-in predicates or operators which are assigned to particular system modules. A check is performed during processing of the built-in predicate to determine whether the constraint can be satisfied. If the constraint cannot be satisfied, the predicate will fail. If IF/Prolog is able to immediately ascertain that the constraint is always satisfied, the predicate will succeed. If IF/Prolog cannot immediately ascertain whether the predicate is satisfiable, the constraint will be suspended until sufficient information is available.

Constraints allow you to formulate conditions for one or more variables, which restrict the permissible domain. A constraint may be

- unary, i.e. it applies only to one variable which can assume a value from a particular range or from a set.
- binary, i.e. two variables must satisfy a condition, e.g. they should form a constant sum.
- global, i.e. several variables must satisfy one condition, e.g. variables in a list should all be different or all negative.

The enormous efficiency benefits offered by constraints result from the immediacy of information access for the system; all information derived from preceding data is available to the system at any point in the program execution. Unnecessary searching is minimized and the solution is reached at over the most direct path possible.

When backtracking is used, IF/Prolog automatically resets changes in constraints.

1.1 Big integers and rational numbers

The range of terms which are interpreted as numbers in IF/Prolog has been extended. Big integers are integers requiring more than machine word (32 or 64 bits) to represent them.

Based on this data type, rational numbers have been implemented. In contrast to floating-point numbers, rational numbers permit numerically precise arithmetic without rounding.

You can enter big integers in the same way as small integers, in decimal or hexadecimal form. IF/Prolog outputs them in decimal form. You identify hexadecimal integers by the prefix `0x`.

Example

```
[user] ?- X = 0xffffffff. ↩

X        = 42949672295 ↩

yes
```

On a 32-bit machine, small integers lie in the range `-0x80000000` through `0x7fffffff` (-2147 483 648 through +2147 483 647), big integers in the range $-2^{1048576}+1$ through $+2^{1048576}-1$. The biggest integer thus has approximately 300000 decimal places.

You identify rational numbers by the prefix `0r`. You can enter a rational number as an uncanceled fraction. The denominator and numerator are positive big integers and are separated by a slash `/`. IF/Prolog normalizes (cancels) the rational number and outputs it in normal form.

Example

```
[user] ?- X = 0r24/3. ↩

X        = 8 ↩

yes
```

i If you specify a value outside the big integer range for a big integer, or if an arithmetic evaluation produces too large a value, IF/Prolog terminates with the system error `'multiple precision integer too big'`.

1.1.1 Arithmetic operations

Arithmetic operations have been extended with the result that IF/Prolog will, where necessary, convert a term to the more general number range:

	Integers	Rational numbers	Real numbers	
special				general

You can also apply the operations `//`, `max`, `min`, `mod` and `rem` (see *IF/Prolog Reference Manual* [1], predicate `is/2`) to big integers. All other arithmetic functions retain the same result types.

When comparing terms of different types by means of the predicates `@<` and `@>`, the following relations are applicable:

big integer	`@<`	rational number
rational number	`@<`	real number
real number	`@<`	small integer
small integer	`@<`	big integer

All predicates and operators which work with small integers can still only process small integers. If you specify big integer with these predicates and operators, IF/Prolog generates the exception `range_error(integer)`. You can use the functions `minint` and `maxint` to query the limits of the range for small integers. The operations `\/`, `/\`, `>>` and `<<`, which interpret an integer as a bit field, also remain restricted to values between `minint` and `maxint`.

i During rounding of big integers to real numbers an overflow will occur if the domain for the real number is exceeded.

1.1.2 Operator for rational division

The operator `rdiv` allows you to divide rational numbers. `rdiv` has the precedence 400 and is right-associative (xfy).

Example

```
[user] ?- A = 6, B = 10, X is A rdiv B.↩

A       = 6
B       = 10
X       = 0r3/5 ↩

yes

[user] ?- X is 15 rdiv 0r1/15.↩

X       = 225 ↩

yes
```

1.2 General built-in predicates

The following built-in predicates are assigned to the module `system`; consequently you can still call them even if you have not imported any of the constraint modules.

This section provides a tabular overview of the built-in predicates followed by descriptions of the predicates in alphabetical order.

1.2.1 Test for constraint variable

Predicate	Purpose
is_constraint/1	Tests whether constraints are defined for the variable.
is_constraint/2	Queries constraint classes.

1.2.2 Test numbers

The following predicates allow you to test the number type of a term:

Predicate	Purpose
integer/1	Test whether a term is a small or big integer.
rational/1	Test whether a term is a rational number.
rational/3	Reduces a rational number to denominator and numerator.

Test for constraint variable

is_constraint(+*Variable*)
is_constraint(+*Variable*, ?*List*)

The predicates `is_constraint/1/2` succeed if *Variable* has one or more constraints; otherwise they fail.

The predicate `is_constraint/2` unifies *List* with a list of atoms which specify the constraint classes in which *Variable* has constraints.

Arguments

Variable	Term
List	List of atoms

Exceptions

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

type_error(atom)
An element of *List* must be a variable or an atom, but is a term of another type.

Example

```
[user] ?- [user].
> :- import(const_delay).
> :- import(const_linear).
> % A predicate to test if a variable has delayed goals
> % associated with it:
> is_delayed(Var) :-
|       is_constraint(Var, List), member(delay, List).
> end_of_file.
*** consult 'user': loaded in 0.03 sec.

yes

[user] ?- freeze(V, write(gabba)), V = a, is_delayed(V).
gabba
no

[user] ?- freeze(V, write(gabba)), is_delayed(V), V=a.
gabba
```

```
V       = a

yes

[user] ?- freeze(A, B is A + 1), A $= C - 5,
| is_constraint(A), C = 12.

A       = 7
B       = 8
C       = 12

yes

[user] ?- freeze(A, B is A + 1), A $= C - 5, C = 12,
| is_constraint(A).

no

[user] ?- freeze(A, B is A + 1), A $= C - 5,
| is_constraint(A, L), C = 12.

A       = 7
B       = 8
C       = 12
L       = [delay,linear]

yes
```

Test for rational number

rational(@*TestTerm* **)**

The predicate `rational/1` succeeds if *TestTerm* is a rational number (but not an integer), otherwise it fails. This distinction is important because the integers are normally considered to be a subset of the rational numbers.

Arguments

TestTerm	Term

Example

```
[user] ?- rational(0r3/7).

yes

[user] ?- X is 0r1/6 + 0r1/5 + 0r1/4 + 0r1/3 + 0r1/2,
          rational(X).

X       = 0r29/20

yes

[user] ?- rational(7).

no
```

See also

rational/3

Decompose a rational number

rational(+*Number*, ?*Numerator*, ?*Denominator* **)**

The predicate `rational/3` determines the denominator and numerator of a rational number.

If *Number* is a rational number, then *Numerator* is unified with the numerator and *Denominator* with the denominator of this number. If *Number* is an integer, *Numerator* is unified with *Number* and *Denominator* with 1.

Arguments

Number	Rational number or integer
Numerator	Integer
Denominator	Integer

Exceptions

instantiation_error
The argument *Number* must not be a variable, but a variable was specified.

type_error(rational)
The argument *Number* must be a rational number or an integer, but is a term of another type.

type_error(integer)
The argument *Numerator* or *Denominator* must be a variable or an integer, but is a term of another type.

Example

```
[user] ?- rational(0r3/7, Numer, Denom).

Numer   = 3
Denom   = 7

yes

[user] ?- X is 0r1/6 + 0r1/5 + 0r1/4 + 0r1/3 + 0r1/2,
          rational(X, Numer, Denom).

X       = 0r29/20
Numer   = 29
Denom   = 20
```

```
yes

[user] ?- rational(7, Numer, Denom).

Numer   = 7
Denom   = 1

yes
```

See also

rational/1, is/2 (operator `rdiv`)

Chapter 2

Coroutines

Coroutines constitute a separate class of constraints. They enable you to satisfy a goal under data control, i.e. the proof of a goal is initiated by the instantiation of a variable. Coroutines allow you to write Prolog programs in a more declarative fashion.

In accordance with the proof model, Prolog proves subgoals procedurally from left to right. This may result in cases which, from the declarative point of view, are not correct:

```
[user] ?- A \= a, A = b.↩

no

[user] ?- A = b, A \= a.↩

A        = b ↩

yes
```

From the logical viewpoint, the order of subgoals is equivalent with respect to satisfiability. Using `freeze/2`, the built-in predicate of the coroutine constraint class, you can delay proving the first goal (A \= a) until the variable *A* is instantiated with b:

```
[user] ?- freeze(A, A \= a), A = b.↩

A        = b ↩

yes
```

Coroutines are inadequate as a general constraint mechanism since, as is the case with standard Prolog, they test consistency conditions passively, i.e. not until a variable has been instantiated. The more efficient method is to restrict the domain of a variable in advance. In this way, the consistency conditions are handled actively, i.e. on instantiation of the variables. Coroutines, however, allow you to define more complex constraints.

The built-in predicate `freeze/2` for the coroutine constraint class is contained in the module `const_delay`. Use the `import` directive to import the module into your database.

```
[user] ?- [user].↩
> :- import(const_delay).↩
> end_of_file.↩
*** consult 'user': loaded in 0.02 sec.

yes
```

i In order to work with coroutines you must always import the module `const_delay`.

Example

The Prolog negation (`\+/1`) succeeds if a goal cannot be proved (negation by failure). Data-controlled proof allows you to achieve a logically correct negation.

```
not_log(Goal) :- deep_freeze(Goal, \+ Goal).

/* Goal is not proved until all variables are ground.*/

deep_freeze(Args, Goal) :-
        var_extract(Args, [], VarList), /* Filter out variables */
        list_freeze(VarList, Goal).     /* Freeze goal          */

list_freeze([], Goal) :- Goal.          /* Prove goal           */

list_freeze([Var|T], Goal) :-
        freeze(Var, deep_freeze([Var|T], Goal)).

var_extract(V, VL, VL) :-
        var(V),
        member(VV, VL),                 /* Variable already exists */
        V == VV, !.

var_extract(V, VL, [V|VL]) :-
        var(V), !.                      /* New variable         */

var_extract(V, VL, VL) :-
        ground(V), !.                   /* Ground               */

var_extract([E|T], VL, VL1) :-
        !,
        var_extract(E, VL, VL0),        /* Test for list        */
        var_extract(T, VL0, VL1).
var_extract(S, VL, VL1) :-
        compound(S), !,                 /* Test for structure   */
        S =.. [_|Args],
        var_extract(Args, VL, VL1).
```

Once you have reconsulted the predicates, the following dialog is possible:

```
[user] ?- \+(member(A, [3,4])), A = 5. ↩

no

[user] ?- not_log(member(A, [3,4])), A = 5. ↩

A       = 5 ↩

yes
```

2.1 Built-in predicates

The following built-in predicates are assigned to the module `const_delay`; consequently you can call them only if you have imported this module.

Delay goal proving

conjunctive_freeze(+*List*, ?*Goal*) `[` @ +*Module* `]`

The predicate `conjunctive_freeze/2` delays the proof of *Goal* until all elements in *List* have been instantiated.

When all elements of *List* are instantiated, the frozen goal is woken up and inserted in the proof tree following the subgoal that caused the instantiation.

Goal can be a composite goal, e.g. a conjunction. In this case, the argument, consisting of subgoals separated by commas, must be enclosed in parenthesis.

Arguments

List	List of terms
Goal	Goal

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `conjunctive_freeze/2` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- conjunctive_freeze([A,B,C], Sum is A+B+C),
A is 11, C is 20, B is (C / A + 3.14), X is Sum - 5.2.

A       = 11
B       = 4.95818
C       = 20
Sum     = 35.9582
X       = 30.7582
```

Compatibility

V5.0B   The predicate `conjunctive_freeze/2` is new.

See also

freeze/2, disjunctive_freeze/2

Term unequality

dif(+*Term1*, +*Term2*)

The predicate `dif/2` succeeds, if *Term1* and *Term2* are not unifiable (Prolog predicate `=/2`). The predicate fails, if the terms are equal (Prolog predicate `==/2`). Otherwise, the terms are unifiable but not equal. Since it cannot be immediately determined, if the terms will always be unequal, the predicate is **suspended**. The test is **resumed** as soon as one variable in either of the terms is instantiated.

If the terms are not unifiable after the instantiation, the predicate that caused the instantiation succeeds. If the terms are equal after the instantiation, the predicate that caused the instantiation fails. Otherwise, the terms are still unifiable but not equal, and the test is suspended again.

Arguments

Term1	Term
Term2	Term

Example

```
[user] ?- dif(A, B), A = a(1), B = a(2).

A       = a(1)
B       = a(2)

yes

[user] ?- A = B, dif(a(A), a(B)).

no

[user] ?- dif(A, B), A = t(A1), B = t(B1), A1 = a, B1 = b.

A       = t(a)
B       = t(b)
A1      = a
B1      = b

yes
```

Compatibility

V5.0B The predicate `dif/2` is new.

Delay goal proving

disjunctive_freeze(+*List*, ?*Goal*) `[ @` +*Module* `]`

The predicate `disjunctive_freeze/2` delays the proof of *Goal* until one of the elements in *List* has been instantiated.

When one element of *List* is instantiated, the frozen goal is woken up and inserted in the proof tree following the subgoal that caused the instantiation. The goal is activated only once, even if more elements of *List* become instantiated later on.

Goal can be a composite goal, e.g. a conjunction. In this case, the argument, consisting of subgoals separated by commas, must be enclosed in parenthesis.

Arguments

List	List of terms
Goal	Goal

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `disjunctive_freeze/2` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- disjunctive_freeze([A,B], current_prolog_flag(A,B)),
B = on.

A       = char_conversion
B       = on ;

A       = signal
B       = on ;

A       = warnings
B       = on ;

no
```

Compatibility

V5.0B The predicate `disjunctive_freeze/2` is new.

See also

freeze/2, conjunctive_freeze/2

Delay goal proving

freeze(+*Variable*, ?*Goal*) `[ @` +*Module* `]`

The predicate `freeze/2` delays the proof of *Goal* until the *Variable* has been instantiated.

When the *Variable* is instantiated, the frozen goal is woken up and inserted in the proof tree following the subgoal that caused the instantiation.

Goal can be a composite goal, e.g. a conjunction. In this case, the argument, consisting of subgoals separated by commas, must be enclosed in parenthesis.

Arguments

Variable	Term
Goal	Goal

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `freeze/2` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

The predicate `print_var/2` enables you to output the instantiations of a variable.

```
print_var(Info, Var) :-
        freeze(Var, (write(Info), write(':'), write(Var),nl).

print_var(Info, Var) :-
        write('Backtrack: '),
        write(Info),
        write (':'),
        write(Var),
        nl,
        fail.

[user] ?- print_var('now', A), write('between'), nl , A = 3.

between

A       = 3 ;
Backtrack: now : _105

no
```

See also

conjunctive_freeze/2, disjunctive_freeze/2

Chapter 3

Linear constraints

Linear constraints are conditions applying to linear domain variables. The linear domain comprises big integers and rational numbers, and is infinite. Using linear constraints, it is possible to solve **non-discrete** problems. Non-discrete means that there are an infinite number of points under which a solution is sought in the search area. These points are represented as rational numbers.

Linear constraints are a set of equations and inequations. A network of constraints arises between the variables which you specify in the equations. With each new equation or inequation, IF/Prolog attempts to solve the equation system with an incremental algorithm.

If you wish to work with linear constraints, you must import the module `const_linear`, which contains the built-in predicates for the linear constraints. Use the `import` directive to import the module into your database:

```
[user] ?- [user].↩
> :- import(const_linear).↩
> end_of_file.↩
*** consult 'user': loaded in 0.03 sec.

yes
```

i You can work with constraints only if you have configured the rational number support and the constraints support at installation.

3.1 Linear terms

Big integers and rational numbers form the basis for linear constraints. Based on these you use linear terms to formulate a linear constraint. A linear term is a rational number, or an arithmetic conjunction of rational numbers. Integers, both big and small, constitute a subset of the rational numbers. The syntax of a linear constraint looks like this:

$$\textit{linear constraint} ::= \textit{linear term} \left\{ \begin{array}{l} \$= \\ \$\backslash= \\ \$< \\ \$=< \\ \$> \\ \$>= \end{array} \right\} \textit{linear term}$$

$$\textit{linear term} ::= \left\{ \begin{array}{l} \textit{variable} \\ \textit{rational constant} \\ +\textit{linear term} \\ -\textit{linear term} \\ \textit{linear term} \left\{ \begin{array}{c} + \\ - \\ \star \end{array} \right\} \textit{linear term} \\ \textit{linear term} \left\{ \begin{array}{c} / \\ \textit{rdiv} \end{array} \right\} \textit{rational constant} \\ (\textit{linear term}) \end{array} \right\}$$

$$\textit{rational constant} ::= \left\{ \begin{array}{l} \textit{integer} \\ \textit{rational number} \\ +\textit{rational constant} \\ -\textit{rational constant} \\ \textit{rational constant} \left\{ \begin{array}{c} + \\ - \\ \star \\ / \\ \textit{rdiv} \end{array} \right\} \textit{rational constant} \\ (\textit{rational constant}) \end{array} \right\}$$

A linear constraint is thus an equation or inequation with two linear terms as operands. Each linear term, which may contain any number of variables, must be reducible to a rational number during the course of processing. This permits efficient processing of the relations.

If an operand fails to comply with the syntax rules, IF/Prolog generates the exceptions `domain_error(linear_term,...)`. The following goals are declaratively not equal, consequently an exception is generated in the second case.

```
[user] ?- A $= 6, A = a.↩

no

[user] ?- A = a, A $= 6.↩

*** E X C E P T I O N: domain_error(linear_term,a)
>>> goal = const_linear : (a $= 6)}

no
```

Multiplication of two variables in linear terms is allowed, but division is not permitted. With multiplication, IF/Prolog delays the proof until one of the variables has been instantiated, thereby making the term linear again. You can achieve such a delay explicitly by using the predicate `linear_if/2/3`. This predicate works in a similar fashion to `freeze/2` in the sense that it is necessary to determine the satisfiability of a condition before Prolog can prove the goal.

Example

A farmer can grow wheat or corn on his field. Each produces a different yield per unit of area of land but also requires a different amount of time for its care; there is also a limit to the maximum amount of work time. The following query to IF/Prolog calculates the maximum yield.

```
[user] ?- Area = 100,                % total available area
| WorkTime = 40,                     % maximum work time
| OutputCorn = 0r5/2,                % corn yield per unit area
| OutputWheat = 0r7/2,               % yield per unit area
| CostsCorn = 0r1/3,                 % time spent per unit area for corn
| CostsWheat = 0r2/3,                % time spent per unit area for wheat
| all_positive([AreaCorn, AreaWheat]),
| Area $>= AreaCorn + AreaWheat,
| WorkTime $>= CostsCorn * AreaCorn + CostsWheat * AreaWheat,
| Output $= AreaCorn * OutputCorn + AreaWheat * OutputWheat,
| linear_maximize(Output, MaxOutput).

Area            = 100
WorkTime        = 40
OutputCorn      = 0r5/2
OutputWheat     = 0r7/2
CostsCorn       = 0r1/3
CostsWheat      = 0r2/3
AreaCorn        = 80
AreaWheat       = 20
Output          = 270
MaxOutput       = 270

yes
```

3.2 Built-in predicates and operators

The following built-in predicates are assigned to the module `const_linear`; consequently you can call them only if you have imported this module.

This section provides a tabular overview of the built-in predicates followed by descriptions of the predicates in alphabetical order.

3.2.1 Linear operators

The built-in operators have been extended to include the linear operators. The operators generate a constraint which is satisfiable when the linear terms on both sides match in the specified relation. The prefix $ identifies the new operators as linear relational operators.

Operator	**Precedence**	**Typ**	**Meaning**
`$=`	700	xfx	equal
`$\=`	700	xfx	not equal
`$<`	700	xfx	less than
`$=<`	700	xfx	less than or equal
`$>`	700	xfx	greater than
`$>=`	700	xfx	greater than or equal

3.2.2 Constraints for list elements

Predicate	**Purpose**
all_different/1	All list elements must be different.
all_negative/1	All list elements must be non-positive.
all_positive/1	All list elements must be non-negative.

3.2.3 Delaying proof of a goal

Conditional execution can delay the proof of a goal until sufficient information is obtained from other program elements. A linear constraint is specified as the condition. Once the satisfiability of this condition has been determined a goal dependent on the condition will automatically be proved.

Predicate	**Purpose**
linear_if/2/3	Delay goal proving until satisfiability of a condition can be determined.

3.2.4 Optimization

Linear terms can be regarded as goal functions. The following predicates can be used to minimize or maximize the value of a goal function under the applicable constraints.

Predicate	**Purpose**
linear_maximize/2	Maximize value of goal function.
linear_minimize/2	Minimize value of goal function.

Linear equations and inequations

?*LinearTerm1* **$=** ?*LinearTerm2*
?*LinearTerm1* **$\=** ?*LinearTerm2*
?*LinearTerm1* **$<** ?*LinearTerm2*
?*LinearTerm1* **$=<** ?*LinearTerm2*
?*LinearTerm1* **$>** ?*LinearTerm2*
?*LinearTerm1* **$>=** ?*LinearTerm2*

The predicates `$=/2`, `$\=/2`, `$</2`, `$=</2`, `$>/2` and `$>=/2` generate linear constraints between the variables in the *LinearTerms*. The constraint is satisfiable, if the linear expressions are in the specified relation to each other.

The multiplication of two variables is delayed until one of them has been instantiated. For example, the subgoal

..., $X * Y$ `$=` Z, ...

will be delayed until either X or Y is bound to a rational constant. When it happens, the corresponding new linear constraint (e.g. $X * C$ `$=` Z) is added to the system.

Arguments

LinearTerm1	Linear term
LinearTerm2	Linear term

Exceptions

domain_error(linear_term)
The argument *LinearTerm1* or *LinearTerm2* must be a linear term. However, it contains non-linear operations.

Example

```
[user] ?- 2*X + 3*Y $= 28, 6*X - 4*Y $= 6.

X       = 5
Y       = 6

yes

[user] ?- [user].
> circle(Circle, Radius) :-
|       Pi is 0r355/113,
|       Circle $= 2 * Pi * Radius.
> end_of_file.
*** consult 'user': loaded in 0.00 sec.

yes

[user] ?- circle(U, 5).

U       = 0r3550/113

yes

[user] ?- circle(10, R).

R       = 0r113/71

yes
```

Constraint for different list elements

all_different(+*List*)

The predicate `all_different/1` generates a constraint whereby all elements of the *List* are different rational numbers or may be instantiated only with such values. The constraint is **not** satisfied if

- a value in *List* is not a rational number.
- a variable in *List* is instantiated with a value which is not a rational number.
- two or more elements in *List* are the same.
- two or more elements in *List* are instantiated with the same value.

Arguments

List	List of rational numbers or linear variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

See also

$\=/2

Constraint for non-positive list elements

all_negative(+*List*)

The predicate `all_negative/1` generates a constraint whereby all elements of the *List* must be rational numbers less than or equal to 0 or may be instantiated only with such values. The constraint is **not** satisfied if

- a value in *List* is a positive rational number.
- a variable in *List* can be instantiated only with a positive rational number.
- a variable in *List* is instantiated with a positive rational number.

Arguments

List	List of rational numbers or linear variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

Example

```
[user] ?- A $>= 0, all_negative([A]).

A       = 0

yes

[user] ?- all_negative([A]), A $> 1.

no
```

See also

all_positive/1

Constraint for non-negative list elements

all_positive(+*List*)

The predicate `all_positive/1` generates a constraint whereby all elements of the *List* must be rational numbers greater than or equal to 0 or may be instantiated only with such values. The constraint is **not** satisfied if

- a value in *List* is a negative rational number.
- a variable in *List* can be instantiated only with a negative rational number.
- a variable in *List* is instantiated with a negative rational number.

Arguments

List	List of rational numbers or linear variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

Example

```
[user] ?- A $=< 0, all_positive([A]).

A       = 0

yes

[user] ?- all_positive([A]), A $< -1.

no
```

See also

all_negative/1

Conditional execution of a goal

linear_if(@*LinearCondition*, ?*ThenGoal*) [@ +*Module*]
linear_if(@*LinearCondition*, ?*ThenGoal*, ?*ElseGoal*) [@ +*Module*]

The predicates `linear_if/2/3` are used to call a goal conditionally. If the constraint *LinearCondition* is satisfied, *ThenGoal* is called. If the condition is not satisfiable, either `true` (`linear_if/2`) or *ElseGoal* (`linear_if/3`) is called. The execution of the predicate is delayed as long as the satisfiability of *LinearCondition* has not been determined.

It cannot be guaranteed that `linear_if/2/3` will be activated immediately when it becomes possible to decide the satisfiability of the condition. For example, a decision on the satisfiability of the condition is delayed in the following situation:

```
linear_if(A $\= B, ThenGoal, ElseGoal), A $\= B, ...
```

In the following situations, however, IF/Prolog has decided that the condition can be satisfied:

```
linear_if(A $\= B, ThenGoal, ElseGoal), A $> B, ...

linear_if(A $\= B, ThenGoal, ElseGoal), A $< 3, B $>= 5, ...
```

Arguments

LinearCondition	Linear constraint
ThenGoal	Goal
ElseGoal	Goal

Hints

The predicate differs from `->/2` by delaying the execution of *ThenGoal* or *ElseGoal* until the satisfiability of *LinearCondition* has been determined. The other difference is that `linear_if/2` succeeds, if *LinearCondition* is not satisfied.

The predicates `linear_if/2/3` are metapredicates and call their goals in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Exceptions

instantiation_error
The argument *LinearCondition* must not be a variable, but a variable was specified.

domain_error(linear_constraint)
The argument *LinearCondition* must be a linear constraint. However, it contains non-linear operations.

instantiation_error
The argument *ThenGoal* or *ElseGoal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *ThenGoal* or *ElseGoal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *ThenGoal* or *ElseGoal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *ThenGoal* or *ElseGoal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *ThenGoal* or *ElseGoal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

See also

$=/2, $\=/2, $</2, $=</2, $>/2, $>=/2, `->/2`

Linear optimization

linear_maximize(?*LinearTerm*, ?*Maximum*)
linear_minimize(?*LinearTerm*, ?*Minimum*)

The predicates `linear_maximize/2` and `linear_minimize/2` search for a maximum or minimum for the given linear term *LinearTerm*, taking consideration the currently applicable constraints, and unify this value with *Maximum* or *Minimum*. If no maximum or minimum satisfying the constraints is found, the predicate fails.
If the current constraints do not delimit a maximum or minimum, then the exception

`evaluation_error(unbounded_solution)`

is generated.

Arguments

LinearTerm	Linear term
Maximum	Rational number
Minimum	Rational number

Exceptions

domain_error(linear_term)
The argument *LinearTerm* must be a linear term. However, it contains non-linear operations.

evaluation_error(unbounded_solution)
The currently applicable constraints do not delimit a domain; consequently no *Minimum* or *Maximum* can be determined.

Example

```
[user] ?- L = [X1, X2, X3, X4, X5, X6, X7, X8],
| all_positive(L),
| 2*X1 - X3 + 3*X4 + X8 $= 1,
| X2 + 3*X4 $= 2,
| 2*X1 + X7 $= 6,
| X3 + X6 $= 6,
| -2*X1 + X3 - 3*X4 + X5 $= 2,
| linear_minimize(4*X1 - 6*X3 + 9*X4 + 66, Min).

L       = [2,2,6,0,0,0,2,3]
X1      = 2
X2      = 2
```

```
X3      = 6
X4      = 0
X5      = 0
X6      = 0
X7      = 2
X8      = 3
Min     = 38

yes

[user] ?- L = [X1, X2, X3, X4, X5, X6, X7, X8],
| all_positive(L),
| 2*X1 - X3 + 3*X4 + X8 $= 1,
| X2 + 3*X4 $= 2,
| 2*X1 + X7 $= 6,
| X3 + X6 $= 6,
| -2*X1 + X3 - 3*X4 + X5 $= 2,
| linear_maximize(4*X1 - 6*X3 + 9*X4 + 66, Max).

L       = [0,1,0,0r1/3,3,6,6,0]
X1      = 0
X2      = 1
X3      = 0
X4      = 0r1/3
X5      = 3
X6      = 6
X7      = 6
X8      = 0
Max     = 69

yes

[user] ?- X $> 4, linear_minimize(X, Min).

no

[user] ?- X $> 4, linear_maximize(X, Max).

*** E X C E P T I O N: evaluation_error(unbounded_solution)
>>> const_linear : linear_maximize(_146,_147)

no

[user] ?- 2*X1 + 2*X2 - X3 $=< 10,
| 3*X1 - 2*X2 + X3 $=< 10,
| X1 - 3*X2 + X3 $=< 10,
| X1 $>= 0,
| X2 $>= 0,
| X3 $>= 0,
| Min $\= -10,
| linear_minimize(X1 + 3*X2 - X3, Min).

no
```

Chapter 4

Constraints for finite domains

Constraints for finite domains are conditions for a variable in a finite domain. A finite domain may contain only certain integers which you can specify directly as intervals, enumerations or sequences. A finite number of instantiation possibilities result from this finite domain. By analogy, it is possible, using constraints for finite domains, to solve **discrete** problems. Discrete means that only a finite number of values satisfy the conditions.

Within the class of constraints for finite domains, you can specify arithmetic and symbolic constraints between variables. You generate arithmetic constraints by using the arithmetic relational operators, and symbolic constraints by using the built-in predicates. The domain variables in the argument(s) should satisfy the generated constraint.

If you wish to work with constraints for finite domains, you must import the module `const_-domain` which contains the built-in predicates for these constraints. Use the `import` directive to import the module into your database:

```
[user] ?- [user].↵
> :- import(const_domain).↵
> end_of_file.↵
*** consult 'user': loaded in 0.02 sec.

yes
```

i You can work with constraints only if you have configured the rational number support and the constraint support at installation.

4.1 Domain variables

As mentioned above, there are different ways in which you can specify finite domains. From the declarative point of view, you specify the same domains with each domain definition; only the respective memory requirement is different:

- For an **interval**, only the domain limits are stored. You specify an interval as follows:
  Variable `in` *Min*`:`*Max*
- For an **enumeration**, all permissible values are stored explicitly. You specify an enumeration as follows:
  Variable `in [` *V1, V2, V3, ..., Vn* `]`
- For a **sequence**, the permissible values are stored in a bit array. You specify a sequence as follows:
  Variable `in` *Min*`..`*Max*

Variable is a term or a list of terms and is also referred to as a domain variable. The type of a domain variable can change during goal proving as a result of the constraints.

Min and *Max* are constant integers. A finite domain may only contain numbers in the range -134217728 through +134217727. A number in this domain is also referred to as a domain integer. If the domain is exceeded, IF/Prolog generates the exception

`domain_error(domain_integer).`

A domain variable can be unified only with a domain integer.

Since a domain integer can also be regarded as a variable with only a single value in its domain, the concept of domain variables also includes domain integers. The exception

`type_error(domain_variable)`

is therefore extended to include domain integers.

Unification between a domain variable and an integer will succeed if the number lies in the current domain for the variable. If a domain variable is unified with a normal variable, the normal variable is bound to the domain variable. If two domain variables are unified, an intersection of the domains is first formed. If this intersection is empty, the unification will fail. If the intersection consists of only one number, both variables are instantiated with this value. Otherwise, one of the domain variables is bound to the second variable. Their domain is then the intersection.

4.2 Arithmetic constraints

Arithmetic constraints are a set of equations and inequations. A network of constraints thus arises between the variables which you specify in the equations. The operands of arithmetic constraints are domain terms. A domain term is a domain integer, or an arithmetic conjunction of domain integers. The syntax of a finite constraint looks like this:

$$\textit{arithmetic constraint} ::= \textit{domain term} \left\{ \begin{array}{l} ?= \\ ?\backslash= \\ ?< \\ ?=< \\ ?> \\ ?>= \end{array} \right\} \textit{domain term}$$

$$\textit{domain term} ::= \left\{ \begin{array}{l} \textit{variable} \\ \textit{domain integer} \\ +\textit{domain term} \\ -\textit{domain term} \\ \textit{domain term} \left\{ \begin{array}{c} + \\ - \\ \star \end{array} \right\} \textit{domain term} \\ (\textit{domain term}) \end{array} \right\}$$

Each domain term, which may contain any number of variables, must be reducible to a domain integer during the course of processing. This permits efficient processing of the relations. If an operand fails to comply with the syntax rules, IF/Prolog generates the exception

```
domain_error(linear_term,...).
```

During the processing of equations and inequations, reasoning over intervals takes place, i.e. between the limits of the domain (minimum, maximum). This allows IF/Prolog to deduce whether the relation is (still) satisfiable, or whether it can exclude values from the domain of a variable.

For reasons of efficiency, it is not possible to perform a general overflow test. You should therefore restrict the domains right at the outset so as to avoid overflow errors.

Example

```
time_conversion(Time, Hour, Min) :-
        Min in 0 : 59,
        Hour in 0 : 23,
        Time ?= Hour * 60 + Min.
```

IF/Prolog defines the minimum and maximum values of the right and left sides so as to determine the satisfiability of the equation. If the variable *Hour* were not limited in this example, it would have a maximum value of 134217727. This number multiplied by 60 would cause an overflow resulting in undefined behavior.

Only linear terms can be used as operands for constraint relations. As an exception, two variables may be multiplied by each other, but the multiplication is delayed until one of the variables has been instantiated. Division of domain terms is not permitted.

You can achieve a delay explicitly by using the predicate `domain_if/2/3`. This predicate works in a similar fashion to `freeze/2` in the sense that it is necessary to determine the satisfiability of the condition before IF/Prolog can prove the goal.

Example

Use the constraints for finite domains to solve the following puzzle:

Five men of different nationalities live in the first five houses of a street. They have different professions and different pets, and drink different drinks. The houses of the five men are painted in different colors:

```
 1.  The Englishman lives in the red house.
 2.  The Spaniard has a dog.
 3.  The Japanese is a painter.
 4.  The Italian drinks tea.
 5.  The Norwegian lives in the first house on the left.
 6.  The occupier of the green house drinks coffee.
 7.  The green house stands next to the white house.
 8.  The sculptor breeds snakes.
 9.  The diplomat lives in the yellow house.
10.  The occupier of the middle house drinks milk.
11.  The Norwegian's house stands immediately to the right of the
     blue house.
12.  The violinist drinks fruit juice.
13.  The house next to the doctor's has a fox.
14.  The house next to the diplomat's has a horse.
```

And the question is: who owns the zebra and who drinks water?

The following arbitrary table is provided to assist in formulating the problem using constraints:

House/ Variables	1	2	3	4	5
N ation C olor P rofession A nimal D rink	England green painter dog juice	Spain red diplomat zebra water	Japan yellow violinist fox tea	Italy blue doctor snails coffee	Norway white sculptor horse milk

Solving of the puzzle requires 25 domain variables, five each for the different nationalities (N1 - N5), colors (C1 - C5) etc. with a domain between 1 and 5. The values represent the house from left to right. According to this table, England corresponds to the variable N1. By analogy with the table, the statements can now be converted into arithmetic constraints:

```
 1.  N1 ?= C2,
 2.  N2 ?= A1,
 3.  N3 ?= P1,
 4.  N4 ?= D3,
 5.  N5 = 1,
 6.  C1 ?= D4,
 7.  C1 ?= C5 + 1,
 8.  P5 ?= A4,
 9.  P2 ?= C3,
10.  D5 = 3,
11.  (N5 ?= C4 -1 ; N5 ?= C4 + 1),
12.  P3 ?= D1,
13.  (A3 ?= P4 - 1 ; A3 ?= P4 + 1),
14.  (A5 ?= P2 - 1 ; A5 ?= P2 + 1).
```

The following query will solve the puzzle:

```
[user] ?- L = [N1,N2,N3,N4,N5,C1,C2,C3,C4,C5,P1,P2,P3,P4,P5,
| A1,A2,A3,A4,A5,D1,D2,D3,D4,D5],
| L in 1..5,
| N1 ?= C2,
| N2 ?= A1,
| N3 ?= P1,
| N4 ?= D3,
| N5 = 1,
| C1 ?= D4,
| C1 ?= C5 + 1,
| P5 ?= A4,
| P2 ?= C3,
| D5 = 3,
| (N5 ?= C4 -1 ; N5 ?= C4 + 1),
| P3 ?= D1,
| (A3 ?= P4 - 1 ; A3 ?= P4 + 1),
| (A5 ?= P2 - 1 ; A5 ?= P2 + 1),
| all_distinct([C1, C2, C3, C4, C5]),
| all_distinct([N1, N2, N3, N4, N5]),
| all_distinct([P1, P2, P3, P4, P5]),
| all_distinct([A1, A2, A3, A4, A5]),
| all_distinct([D1, D2, D3, D4, D5]),
| label(L).

L       = [3,4,5,2,1,5,3,1,2,4,5,1,4,2,3,4,5,1,3,2,4,1,2,5,3]
N1      = 3
N2      = 4
N3      = 5
N4      = 2
N5      = 1
C1      = 5
C2      = 3
C3      = 1
```

```
C4      = 2
C5      = 4
P1      = 5
P2      = 1
P3      = 4
P4      = 2
P5      = 3
A1      = 4
A2      = 5
A3      = 1
A4      = 3
A5      = 2
D1      = 4
D2      = 1
D3      = 2
D4      = 5
D5      = 3

yes
```

The table now looks like this:

Houses	1	2	3	4	5
N ation	Norway	Italy	England	Spain	Japan
C olor	yellow	blue	red	white	green
P rofession	diplomat	doctor	sculptor	violinist	painter
A nimal	fox	horse	snails	dog	zebra
D rink	water	tea	milk	juice	coffee

The zebra thus belongs to the Japanese and the Norwegian drinks water.

4.3 Built-in operators and predicates

The following built-in predicates are assigned to the module `const_domain`; consequently you can call them only if you have imported this module.

This section provides a tabular overview of the built-in predicates followed by descriptions of the predicates in alphabetical order.

4.3.1 Arithmetic Constraints

The built-in operators have been extended to include the relational operators for domains. The operators generate a constraint which is satisfiable when the domain terms on both sides match in the specified relation. The prefix `?` identifies the new operators as relational operators for finite domains. The constraints which you generate with these relational operators are also referred to as arithmetic constraints.

Operator	**Precedence**	**Type**	**Meaning**
?=	700	xfx	equal
?\=	700	xfx	not equal
?<	700	xfx	less than
?=<	700	xfx	less than or equal
?>	700	xfx	greater than
?=<	700	xfx	greater than or equal

4.3.2 Symbolic constraints

Symbolic constraints allow you to express dependencies between domain variables in a simple way. A predicate which generates a symbolic constraint normally contains a list as its argument. The domain variables in this list should satisfy the specified constraint.

Predicate	**Purpose**
all_distinct/1	Constraint for different list elements
atmost/3	Define maximum number of identical values
cardinality/3	Cardinality constraint
cumulative/4	Cumulative constraint
element/3	Constraints for list elements
exactly/3	Define specific number of identical elements in a list
in/2	Limit domain
maximum/2	Constraint for maximum value
minimum/2	Constraint for minimum value

4.3.3 Selection predicates

The selection predicates allow you to generate consistent instantiations for domain variables.

Predicate	**Purpose**
deleteff/3	Select variable
deleteffc/3	Select variable
deleteff0/3	Select variable
indomain/1	Generate value for variable(s)
label/1	Generate values for variable(s)
label/2	Generate values for variable(s)
label_tb/3	Generate values for variable(s)

4.3.4 Non-logical predicates

Non-logical predicates allow you to request information about one or more domain variables.

Predicate	**Purpose**
domain/2	Query domain
domain_maximum/2	Query domain maximum
domain_minimum/2	Query domain minimum
domain_size/2	Query domain size
is_consecutive/1	Test variable type
is_domain/1	Test variable type
is_enumeration/1	Test variable type
is_in_domain/2	Test domain affiliation
is_interval/1	Test variable type
lmaxdomain/2	Query domain limit
lmindomain/2	Query domain limit
lmaxmin/2	Query domain limit
lminmax/2	Query domain limit

4.3.5 Delaying proof of a goal

Conditional execution can delay the proof of a goal until sufficient information is obtained from other program elements. An arithmetic constraint is specified as the condition. Once it has been determined that this condition can be satisfied, a goal dependent on the condition will automatically be proved.

Predicate	**Purpose**
domain_if/2/3	Delay goal proving until satisfiability of a condition can be determined

4.3.6 Optimization

The following predicates optimize a given goal function using the ”branch-and-bound” - method.

Predicate	**Purpose**
minimize_bb/2/5	Optimize result value
minimize_maximum/2/5	Optimize maximum value of a list

Arithmetic equations and inequations

?*DomainTerm1* **?=** **?***DomainTerm2*
?*DomainTerm1* **?\=** **?***DomainTerm2*
?*DomainTerm1* **?<** **?***DomainTerm2*
?*DomainTerm1* **?=<** **?***DomainTerm2*
?*DomainTerm1* **?>** **?***DomainTerm2*
?*DomainTerm1* **?>=** **?***DomainTerm2*

The predicates `?=/2, ?\=/2, ?</2, ?=</2, ?>/2, ?>=/2` generate arithmetic constraints between the variables in the *DomainTerms*. The constraint is satisfiable when the variables in the terms are constrained to such values that the specified relation is satisfiable.

When the arguments are ground, it can be immediately checked whether the condition is true or false. Otherwise, the satisfiability of a condition can be decided by checking the limit values of the domain variables. Each time a limit value of a variable changes, the condition is checked again. It is thus possible to determine the satisfiability of a condition without instantiating the variables. At the latest, the satisfiability is determined when all variables have been instantiated.

Generally the arguments should be linear. However, the multiplication of two variables is allowed, but is delayed until one of them has been instantiated.

Arguments

DomainTerm1	Domain term
DomainTerm2	Domain term

Exceptions

domain_error(linear_term)
The argument *DomainTerm1* or *DomainTerm2* must be a domain term. However, it contains non-linear operations.

type_error(domain_integer)
An element of *DomainTerm1* or *DomainTerm2* must be a domain integer, but is a term of another type.

Example

```
[user] ?- X in 1..4, Y in 3..7, X ?= Y, indomain(Y).

X       = 3
Y       = 3 ;
```

```
X       = 4
Y       = 4 ;

no

[user] ?- X in 1..4, Y in 6:8, X ?= Y.

no

[user] ?- R in 0:1, [E, T] in 0..9, E ?\= 1, T ?\= 1,
| R + E + 1 ?= 10 + T,
| domain(T, DT), domain(E, DE), domain(R, DR).

R       = __1
E       = __2
T       = 0
DT      = [0]
DE      = [8,9]
DR      = [0,1]

yes
```

See also

in/2, domain/2

Generate constraint for distinct list elements

all_distinct(?*List*)

The predicate `all_distinct/1` generates a constraint whereby any pair of elements in *List* are different integers or are instantiated with different integers. The predicate considers only the consistency between any pair of variables, but does not check the global consistency between all variables. For reasons of efficiency, intervals and free variables in *List* are converted into a sequence. Free variables are limited to the range -10000 through +10000.

i If an interval is not sufficiently limited, the memory requirements for the corresponding sequence can be extremely large.

The predicate will fail if an element of *List* is neither a variable nor an integer.

Arguments

List	List of domain variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

Example

```
[user] ?- [A, B] in 1..2, indomain(A), all_distinct([A, B]).

A       = 1
B       = 2 ;

A       = 2
B       = 1 ;

no

[user] ?- [A, B, C] in 1..2, all_distinct([A, B, C]),
| domain(A, DA), domain(B, DB), domain(C, DC).

A       = __1
B       = __2
```

```
C       = __3
DA      = [1,2]
DB      = [1,2]
DC      = [1,2] ;

no

[user] ?- [A, B, C] in 1..2, all_distinct([A, B, C]), label([A, B, C]).

no
```

See also

?\=/2

Constrain minimum or maximum number of identical values

atleast(+*Limit*, ?*List*, +*Value*)

atmost(+*Limit*, ?*List*, +*Value*)

The predicate `atleast/3` generates a constraint between two integers and the elements of the *List*. The constraint is satisfiable if at least *Limit* elements of the *List* can have the same *Value*. If *List* is a free variable, it is instantiated with a list of *Limit* elements having the same *Value*.

The predicate `atmost/3` generates a constraint between two integers and the elements of the *List*. The constraint can be satisfied if no more than *Limit* elements of the *List* have the same *Value*. If *List* is a free variable, it is instantiated with the empty list, independent of the values for *Limit* and *Value*.

Arguments

Limit	Domain integer
List	List of domain variables
Value	Domain integer

Exceptions

instantiation_error
The argument *Limit* or *Value* must not be a variable, but a variable was specified.

type_error(domain_integer)
The argument *Limit* or *Value* must be a domain integer, but is a term of another type.

type_error(list)
The argument *List* must be a variable or a list, but is a term of another type.

Example

You can also define the predicate `atmost/3` yourself using `cardinality/3`:

```
my_atmost(Maximum, List, Value) :-
        collect(List, Value, Conditions),
        cardinality(*, Maximum, Conditions), !.

collect([], _, []).
collect([V|R], Value, [V ?= Value|Conditions]) :-
        collect(R, Value, Conditions).
```

In the following, *Travellers* is a list of domain variables which defines when a number of people might start a journey. *Departure* is the departure time of the train and *Seats* indicates how many people may travel on the train:

```
train_usage(Travellers, Departure, Seats) :-
        atmost(Seats, Travellers, Departure).
```

The predicate `atleast/3` can generate values for domain variables that are not of type `interval`:

```
[user] ?- [A,B,C] in 0..10, atleast(2,[A,B,C],7), B ?\= 7.

A       = 7
B       = __2
C       = 7 ;
```

Compatibility

V5.1A The predicate `atleast/3` is new.

See also

all_distinct/1, cardinality/3, exactly/3

Cardinality constraint

cardinality(?*LowerLimit*, ?*UpperLimit*, +*ConditionList*)

The predicate `cardinality/3` allows you to combine two or more primitive constraints conjunctively (AND) or disjunctively (OR). This produces a non-primitive constraint which applies to a set of constraints. The constraint is satisfied if at least *LowerLimit* and at most *UpperLimit* conditions in the *ConditionList* are satisfied.

Specifying an asterisk $*$ as *LowerLimit* indicates that the minimum is equal to 0, i.e. none of the conditions need be satisfied.

Specifying an asterisk $*$ as *UpperLimit* indicates that the maximum must be equal to the number of conditions in the *ConditionList*, i.e. all the conditions must be satisfied.

Conjunction (AND) of the constraints in the *ConditionList* can be expressed by setting *LowerLimit* and *UpperLimit* equal to the number of constraints:

```
conjunction(ConditionList) :-
        list_length(ConditionList, Number),
        cardinality(Number, Number, ConditionList).
```

Exclusive disjunction (at most one of the constraint may be satisfied) of the constraints in the *ConditionList* can be expressed by setting *LowerLimit* equal to 0 and setting *UpperLimit* equal to 1:

```
disjunction(ConditionList) :-
        cardinality(0, 1, ConditionList).
```

Negation (NOT) of the constraints in the *ConditionList* can be expressed by setting *Lower-Limit* and *UpperLimit* equal to 0:

```
negation(ConditionList) :-
        cardinality(0, 0, ConditionList).
```

Arguments

LowerLimit	Domain variable or the atom $*$
UpperLimit	Domain variable or the atom $*$
ConditionList	A list of arithmetic constraints

Exceptions

instantiation_error
The argument *ConditionList* must not be a variable, but a variable was specified.

type_error(list)
The argument *ConditionList* must be a regular list, but is a term of another type or not regular.

domain_error(linear_constraint)
An element of *ConditionList* must be a linear constraint. However, it contains non-linear operations.

Example

The predicate `atleast_greatereq/3` generates a combined constraint where at least *Minimum* list elements must be greater than or equal to the specified *Value*:

```
atleast_greatereq(Minimum, List, Value) :-
        collect(List, Value, Conditions),
        cardinality(Minimum, *, Conditions).

collect([], _, []).
collect([V|R], Value, [V ?>= Value|Condition]) :-
        collect(R, Value, Condition).

[user] ?- atleast_greatereq(2, [4, 1, 2, 3], 2).

yes

[user] ?- atleast_greatereq(2, [1, 2], 2).

no

[user] ?- cardinality(1, 1, [X ?= 2, Y ?\= 3, Z ?\= 4]),
| (
| X = 2;
| Y = 3, Z = 4;
| Z = 0
| ).

X       = 2
Y       = 3
Z       = 4 ;

X       = 2
```

```
    Y       = 3
    Z       = 4 ;

    X       = __1
    Y       = 3
    Z       = 0 ;

    no
```

See also

atmost/3, exactly/3

Cumulative constraint

cumulative(?*Start*, ?*Duration*, ?*Resource*, +*Limit*)

The predicate `cumulative/4` generates constraints between points of intervals. These intervals are formed from the elements of the lists *Start* and *Duration*. A point *p* lies within the interval *i* if the following applies:

$$\text{Start[i]} \leq \text{p} < \text{Start[i]} + \text{Duration[i]}$$

Each interval is assigned a resource value *Resource[i]*. The resource values of all intervals for a point *p* are totaled. `cumulative/4` generates a constraint which ensures that this total is less than or equal to the resource limit *Limit* in each point. The constraint is satisfied if the maximum resource usage *Limit* is not exceeded in any point.

You should limit the values in the lists *Start* and *Duration* appropriately before activating `cumulative/4`. The predicate `cumulative/4` itself constrains the values in the list *Duration* to be greater than or equal to 0, and in the list *Resource* to be between 0 and *Limit*. The lists *Start*, *Duration* and *Resource* must have the same length.

Hints

`cumulative/4` is well suited for use in scheduling tasks. *Start* then corresponds to the start times of the tasks, *Duration* to the relevant periods of time involved, and *Resource* specifies how many resource units a particular task requires.

Arguments

Start	List of domain variables or domain integers
Duration	List of domain variables or domain integers
Resource	List of variables or domain integers
Limit	Domain integer

Exceptions

instantiation_error
The argument *Start*, *Duration*, *Resource* or *Limit* must not be a variable, but a variable was specified.

type_error(list)
The argument *Start*, *Duration* or *Resource* must be a regular list, but is a term of another type or not regular.

domain_error(lists_with_identical_length)
The arguments *Start*, *Duration* and *Resource* are not all lists of the same length.

type_error(domain_variable)
An element of *Start* or *Duration* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *Start* or *Duration* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(variable)
An element of *Resource* must be a variable or a domain integer, but is a term of another type.

type_error(domain_integer)
The argument *Limit* must be a domain integer, but is a term of another type.

Example

```
[user] ?- cumulative([1,1,2], [2,1,1], [2,1,1], 3).

yes

[user] ?- [S1,S2,S3] in 1:2,
| cumulative([S1,S2,S3], [2,1,1], [2,1,1], 2).

S1      = 2
S2      = 1
S3      = 1 ;

no

[user] ?- [S, D, R] in 1..2,
| cumulative([1,1,S], [2,1,D], [2,1,R], 3),
| domain(D,DL).

S       = 2
D       = __2
R       = 1
DL      = [1,2]

yes

[user] ?- [S, D, R] in 1..2,
| cumulative([1,1,S], [2,D,1], [R, 1,1], 3),
| label([S, D, R]),
| write(job(start=S, duration=D, resource=R)), nl,
| fail.
```

```
job(start = 1,duration = 1,resource = 1)
job(start = 1,duration = 2,resource = 1)
job(start = 2,duration = 1,resource = 1)
job(start = 2,duration = 1,resource = 2)
job(start = 2,duration = 2,resource = 1)

no
```

Compatibility

V5.1A The list length of the arguments is no more limited to 127.

See also

is_domain/1, disjunctive/2

Select variable

deleteff(`-`*Variable*, @*List*, ?*Rest*)
deleteffc(`-`*Variable*, @*List*, ?*Rest*)
deleteff0(`-`*Variable*, @*List*, ?*Rest*)

The predicates `deleteff/3, deleteff0/3` and `deleteffc/3` support the so-called first fail principle. The predicates unify the *Variable* with a domain variable in *List* using different heuristics:

- `deleteff/3` unifies *Variable* with the domain variable in *List* whose domain has the fewest elements. If several domain variables fall under this criterion, the first complying domain variable is used.
- `deleteffc/3` unifies *Variable* with the domain variable in *List* whose domain has the fewest elements. If several domain variables fall under this criterion, the domain variable which occurs in more constraints is used.
- `deleteff0/3` unifies *Variable* with the domain variable in *List* whose domain has the lowest minimum. If the same minimum applies to a number of domain variables, the domain variable which occurs in more constraints is used.

The argument *Rest* is unified with *List* without *Variable*. If domain integers precede the selected *Variable* in *List*, they are not included in *Rest*.

Arguments

Variable	Variable
List	List of domain variables
Rest	List of domain variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- A in 3..10, B in [6, 8, 12], C in -2..8, D in [-1, 1, 3],
| L = [A, B, C, D],
| all_distinct(L),
| element(C, [8, 1, -1, 12, 5, 3], D),
| deleteff(Vff, L, Rff),
| deleteff0(Vf0, L, Rf0),
| deleteffc(Vffc, L, Rffc).

A       = __1
B       = __2
C       = __3
D       = __4
L       = [__1,__2,__3,__4]
Vff     = __2
Rff     = [__1,__3,__4]
Vf0     = __4
Rf0     = [__1,__2,__3]
Vffc    = __3
Rffc    = [__1,__2,__4]

yes
```

See also

label/1/2

Non-overlapping rectangles

diffn(+*RectangleList*)

The predicate `diffn/1` constrains n-dimensional rectangles to be non-overlapping. Each rectangle is defined by its coordinate of origin and length in n dimensions. Thus, a rectangle is a tuple of domain variables or domain integers

$$\textit{Rectangle} ::= \texttt{[}\ O_1, ..., O_n, L_1, ..., L_n\ \texttt{]}$$

The argument *RectangleList* is a list of such tuples.

The constraint is satisfiable, if in some dimension there is a possibility for each rectangle to be either before or after other rectangles. It is satisfied, if there is no possibility of overlapping.

The reasoning for this constraint is based on boundary checking. If the satisfiability can not be decided, the constraint remains suspended, and is reactivated when a boundary of some variable is changed.

Hints

The constraint can be used for multi-dimensional placement tasks that occur in scheduling, cutting and geometrical placement problems.

Arguments

RectangleList	List of *Rectangle*s
Rectangle	List of domain variables

Exceptions

instantiation_error
The argument *RectangleList* must not be a variable, but a variable was specified.

type_error(list)
The argument *RectangleList* or each *Rectangle* must be a list, but is a term of another type.

domain_error(lists_with_identical_length)
The arguments in *RectangleList* are not all lists of the same length.

domain_error(list_of_origins_and_lengths)
Each *Rectangle* must be a list of origins and lengths, i.e. the length of each *Rectangle* list must be even.

type_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a **normal** variable.

Example

Place three segments in such a way that they do not overlap:

```
[user] ?- listing(origins).

% *** user: origins / 1 ***
origins(Origins) :-
    Origins = [O1,O2,O3],
    Origins in 1 .. 100,
    D1 = 3,
    D2 = 5,
    D3 = 2,
    diffn([[O1,D1],[O2,D2],[O3,D3]]),
    label(Origins) .

[user] ?- origins(Origins).

Origins = [1,4,9]

yes
```

Place three rectangles in such a way that they do not overlap:

```
[user] ?- listing(rectangles).

% *** user: rectangles / 1 ***
rectangles(Rectangles) :-
    Rectangles = [[X1,Y1,4,2],[X2,Y2,7,3],[X3,Y3,9,5]],
    [X1,X2,X3] in 1 .. 100,
    [Y1,Y2,Y3] in 1 .. 100,
    diffn([[X1,Y1,4,2],[X2,Y2,7,3],[X3,Y3,9,5]]),
    label([X1,Y1,X2,Y2,X3,Y3]) .

[user] ?- rectangles(Rectangles).

Rectangles      = [[1,1,4,2],[1,3,7,3],[1,6,9,5]]

yes
```

Compatibility

V5.1A The predicate `diffn/1` is new.

See also

in/2, diffn/3/4, all_distinct/1, cumulative/4, disjunctive/2

Non-overlapping rectangles

diffn(+*RectangleList*, +*MinVolume*, +*MaxVolume***)**

The predicate `diffn/3` is an extension of `diffn/1` constraint.

The argument *RectangleList* is a list of *m* tuples:

$$RectangleList ::= \texttt{[}\ Rectangle_1, ..., Rectangle_m\ \texttt{]}$$
$$Rectangle_i ::= \texttt{[}\ O_1, ..., O_n, L_1, ..., L_n\ \texttt{]}$$

The argument *MinVolume* is an atom `unused` or a list of domain integers, specifying the minimum volume of each rectangle:

$$MinVolume ::\text{-}\ \texttt{[}\ Min_1, ..., Min_m\ \texttt{]}$$

If the atom `unused` is given, then the maximum volume is not limited.

The argument *MaxVolume* is an atom `unused` or a list of domain integers, specifying the maximum volume of each rectangle:

$$MaxVolume ::\text{-}\ \texttt{[}\ Max_1, ..., Max_m\ \texttt{]}$$

If the atom `unused` is given, then the maximum volume is not limited.

The constraint is satisfiable if the underlying `diffn/1` constraint is satisfiable and additionally, if the volume of each rectangle is in the given domain.

Hints

The volume constraints can be used in geometrical placement problems, where there are explicit limits for the objects.

Arguments

RectangleList	List of *Rectangle*s
Rectangle	List of domain variables
MinVolume	Atom `unused` or list of domain integers
MaxVolume	Atom `unused` or list of domain integers

Exceptions

instantiation_error
The argument *RectangleList* must not be a variable, but a variable was specified.

type_error(list)
The argument *RectangleList* or each *Rectangle* must be a list, but is a term of another type.

domain_error(lists_with_identical_length)
The arguments in *RectangleList* are not all lists of the same length.

domain_error(list_of_origins_and_lengths)
Each *Rectangle* must be a list of origins and lengths, i.e. the length of each *Rectangle* list must be even.

type_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a **normal** variable.

domain_error(list_of_volumes)
The length of the argument *MinVolume* or *MaxVolume* must be equal to the number of rectangles.

type_error(domain_integer)
An element of *MinVolume* or *MaxVolume* must be a domain integer, but is a term of another type.

Example

Place three rectangles, whose minimum and maximum surface is limited, in such a way that they do not overlap.

```
[user] ?- listing(surfaces).

% *** user: surfaces / 1 ***
surfaces(Sol) :-
    Sol = [[X1,Y1,L1,H1],[X2,Y2,L2,H2],[X3,Y3,L3,H3]],
    [X1,X2,X3] in 1 .. 100,
    [Y1,Y2,Y3] in 1 .. 100,
    [L1,L2,L3] in 1 .. 100,
    [H1,H2,H3] in 1 .. 100,
    diffn([[X1,Y1,L1,H1],[X2,Y2,L2,H2],[X3,Y3,L3,H3]],
        [12,23,14],[15,30,19]),
    label([X1,Y1,L1,H1,X2,Y2,L2,H2,X3,Y3,L3,H3]) .

yes

[user] ?- surfaces(Sol).

Sol     = [[1,1,1,12],[1,13,1,23],[1,36,1,14]]

yes
```

Compatibility

V5.1A The predicate `diffn/3` is new.

See also

in/2, diffn/1/4, all_distinct/1, cumulative/4, disjunctive/2

Non-overlapping rectangles

diffn(+*RectangleList*, +*MinVolume*, +*MaxVolume*, +*End*)

The predicate `diffn/4` is an extension of `diffn/1` and `diffn/3` constraints.

The argument *RectangleList* is a list of *m* tuples:

$$\textit{RectangleList} ::= \texttt{[}\ \textit{Rectangle}_1,\ ...,\ \textit{Rectangle}_m\ \texttt{]}$$
$$\textit{Rectangle}_i ::= \texttt{[}\ O_1,\ ...,\ O_n,\ L_1,\ ...,\ L_n\ \texttt{]}$$

The argument *MinVolume* is an atom `unused` or a list of domain integers, specifying the minimum volume of each rectangle:

$$\textit{MinVolume} ::\text{-}\ \texttt{[}\ Min_1,\ ...,\ Min_m\ \texttt{]}$$

The argument *MaxVolume* is an atom `unused` or a list of domain integers, specifying the maximum volume of each rectangle:

$$\textit{MaxVolume} ::\text{-}\ \texttt{[}\ Max_1,\ ...,\ Max_m\ \texttt{]}$$

The argument *End* is an atom `unused` or a list of domain integers or domain variables:

$$\textit{End} ::\text{-}\ \texttt{[}\ End_1,\ ...,\ End_n\ \texttt{]}$$

The end values specify the domain in which the maximum of the ends of the rectangles must lie. If the atom `unused` is given, then the ends are not limited.

The constraint is satisfiable if the underlying `diffn/3` constraint is satisfiable and additionally, if the end of each rectangle in each dimension is in the given domain.

Hints

The end constraint can be used in placement and scheduling problems, where there is an explicit limit on the placement space or a limit for the general completion of the scheduled tasks.

Arguments

RectangleList	List of *Rectangle*s
Rectangle	List of domain variables
MinVolume	Atom `unused` or list of domain integers
MaxVolume	Atom `unused` or list of domain integers
End	Atom `unused` or list of domain variables or domain integers

Exceptions

instantiation_error

The argument *RectangleList* must not be a variable, but a variable was specified.

type_error(list)
The argument *RectangleList* or each *Rectangle* must be a list, but is a term of another type.

domain_error(lists_with_identical_length)
The arguments in *RectangleList* are not all lists of the same length.

domain_error(list_of_origins_and_lengths)
Each *Rectangle* must be a list of origins and lengths, i.e. the length of each *Rectangle* list must be even.

type_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *Rectangle* must be a domain variable or a domain integer, but is a **normal** variable.

domain_error(list_of_volumes)
The length of the argument *MinVolume* or *MaxVolume* must be equal to the number of rectangles.

type_error(domain_integer)
An element of *MinVolume* or *MaxVolume* must be a domain integer, but is a term of another type.

type_error(domain_variable)
An element of *End* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *End* must be a domain variable or a domain integer, but is a **normal** variable.

Example

Place three rectangles in such a way that they do not overlap, and limit the general end in each dimension:

```
[user] ?- listing(ends).

% *** user: ends / 2 ***
ends(Rect,End) :-
    Rect = [[X1,Y1],[X2,Y2],[X3,Y3]],
    End = [EndX,EndY],
    [X1,X2,X3] in 1 .. 100,
    [Y1,Y2,Y3] in 1 .. 100,
    [EndX,EndY] in 1 .. 100,
    diffn([[X1,Y1,4,2],[X2,Y2,7,3],[X3,Y3,9,5]],
            unused,unused,[EndX,EndY]),
        label([X1,Y1,X2,Y2,X3,Y3]) .

    yes

    [user] ?- ends(Rect, End).

    Rect    = [[1,1],[1,3],[1,6]]
    End     = [10,11]

    yes
```

Compatibility

V5.1A The predicate `diffn/4` is new.

See also

in/2, diffn/1/3, all_distinct/1, cumulative/4, disjunctive/2

Disjunctive constraint

disjunctive(?*Start*, ?*Duration*)

The predicate `disjunctive/2` generates constraints between points of intervals. These intervals are formed from the elements of the lists *Start* and *Duration*. A point *p* lies within the interval *i* if the following applies:

$$\text{Start[i]} \leq \text{p} < \text{Start[i]} + \text{Duration[i]}$$

`disjunctive/2` generates a constraint which ensures that each point lies in at most one interval, i.e. the intervals are disjunct. The constraint is satisfied if no point belongs to more than one interval.

You should limit the values in the lists *Start* and *Duration* appropriately before activating `disjunctive/2`. In addition, the predicate `disjunctive/2` constrains the values in the list *Duration* to be greater than or equal to 0. The lists *Start* and *Duration* must have the same length.

Hints

`disjunctive/2` is well suited for use in scheduling tasks. *Start* then corresponds to the start times of the tasks, *Duration* to the relevant periods of time involved.

Arguments

Start	List of domain variables or domain integers
Duration	List of domain variables or domain integers

Exceptions

instantiation_error
The argument *Start* or *Duration* must not be a variable, but a variable was specified.

type_error(list)
The argument *Start* or *Duration* must be a regular list, but is a term of another type or not regular.

domain_error(lists_with_identical_length)
The arguments *Start* and *Duration* are not all lists of the same length.

type_error(domain_variable)
An element of *Start* or *Duration* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *Start* or *Duration* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- disjunctive([0, 1, 4, 7, 8], [1, 2, 3, 1, 2]).

yes

[user] ?- [S1, S2] in 1..7, D1 = 2, D2 = 4,
| disjunctive([S1, S2], [D1, D2]),
| S1 + D1 ?=< 7, S2 + D2 ?=< 7,
| indomain(S1).

S1       = 1
S2       = 3
D1       = 2
D2       = 4 ;

S1       = 5
S2       = 1
D1       = 2
D2       = 4 ;

no

[user] ?- S = [S1, S2, S3], D = [D1, D2, D3],
| S1 = 1, S2 ?> 0, S3 = 4,
| D1 ?> 0, D2 = 2, D3 ?> 0,
| End = 5,
| S1 + D1 ?=< End, S2 + D2 ?=< End, S3 + D3 ?=< End,
| disjunctive(S, D),
| label(D).

S        = [1,2,4]
S1       = 1
S2       = 2
S3       = 4
D        = [1,2,1]
D1       = 1
D2       = 2
D3       = 1
End      = 5 ;

no
```

Compatibility

V5.1A The list length of the arguments is no more limited to 127.

See also

cumulative/4, is_domain/1

Absolute distance between variables

distance(+*X*, +*Y*, +*Comp*, +*Dist*)

The predicate `distance/4` constrains the absolute distance between two domain variables. The constraint is similar to arithmetic constraints, but is based on the absolute distance between the domain variables.

The argument *Comp* is an atom that specifies the distance relation:

Comp	The distance between *X* and *Y* must be ...
`?=`	exactly *Dist*
`?\=`	different than *Dist*
`?<`	smaller than *Dist*
`?=<`	smaller than or equal to *Dist*
`?>`	greater than *Dist*
`?>=`	greater than or equal to *Dist*

The reasoning for this constraint is based on boundary checking. For variables of **enumeration** and **sequence** type, it can remove values from the middle of the domain. For variables of **interval** type, only boundaries can be immediately modified. If the satisfiability can not be decided, the constraint remains suspended, and is reactivated when a boundary of either variable is changed.

The constrain helps to avoid disjunction especially in cases when distance must be equal to or greater than a give value (see example below).

Arguments

X	Domain variable
Y	Domain variable
Comp	Comparison
Dist	Domain integer

Exceptions

type_error(domain_variable)
The argument *X* or *Y* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *X* or *Y* must be a domain variable or a domain integer, but is a **normal** variable.

domain_error(comparison)
The argument *Comp* must be a comparison operator.

type_error(domain_integer)
The argument *Dist* must be a domain integer, but is a term of another type.

Example

We want the distance of two variables to be exactly 5:

```
?- [X, Y] in 0..20, ( X - Y ?= 5; Y - X ?= 5 ), X = 15.

X       = 15
Y       = 10 ;

X       = 15
Y       = 20 ;

no
```

The above disjunction can be expressed more elegantly with `distance/4`:

```
?- [X, Y] in 0..20, distance(X, Y, ?=, 5),
   X = 15, domain(Y, DY).

X       = 15
Y       = __2
DY      = [10,20]

yes

?- [A, B] in 1..30, [X, Y] in 1..10,
   2 * X ?= A, 3 * Y ?= B,
   distance(A, B, ?>, 26).

A       = 2
B       = 30
X       = 1
Y       = 10

yes

?- [A, B] in 2:5, distance(A, B, ?>, 1), label([A,B]).

A       = 2
B       = 4 ;
```

```
A       = 2
B       = 5 ;

A       = 3
B       = 5 ;

A       = 4
B       = 2 ;

A       = 5
B       = 2 ;

A       = 5
B       = 3 ;

no
```

Compatibility

V5.1A The predicate `distance/4` is new.

See also

in/2, ?\=/2, indomain/1/2, notin/3

Query domain

domain(+*Variable*, ?*List*)

The predicate `domain/2` unifies *List* with the values that lie in the domain of *Variable*. If *Variable* is a domain integer, *List* is unified with a list containing only the element *Variable*.

i If *Variable* is not sufficiently constrained, the list can become extremely long.

Arguments

Variable	Domain variable
List	List of domain integers

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- X in [3, 5, 7, 9], domain(X, L1),
| X ?\= 7, domain(X, L2),
| X = 9, domain(X, L3).

X       = 9
L1      = [3,5,7,9]
L2      = [3,5,9]
L3      = [9]

yes

[user] ?- X ?> 5,
| domain_minimum(X, Min), domain_maximum(X, Max),
| % domain(X, L) would create a list with S elements!
| domain_size(X, S).

X       = __1
```

```
Min     = 6
Max     = 134217727
S       = 134217722

yes
```

See also

indomain/1/2, domain_size/2

Conditional execution of a goal

domain_if(@*Condition*, +*ThenGoal*) `[ @ +`*Module*` ]`
domain_if(@*Condition*, +*ThenGoal*, +*ElseGoal*) `[ @ +`*Module*` ]`

The predicates `domain_if/2/3` are used to call a goal conditionally. If the arithmetic constraint *Condition* is satisfied, *ThenGoal* is called. If the constraint is not satisfiable, either `true` (`domain_if/2`) or *ElseGoal* (`domain_if/3`) is called. The execution of the predicate is delayed as long as the satisfiability of *Condition* has not been determined.

Arguments

Condition	Arithmetic constraint
ThenGoal	Goal
ElseGoal	Goal

Exceptions

instantiation_error
The argument *Condition* must not be a variable, but a variable was specified.

domain_error(linear_constraint)
The argument *Condition* must be a linear constraint. However, it contains non-linear operations.

instantiation_error
The argument *ThenGoal* or *ElseGoal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *ThenGoal* or *ElseGoal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *ThenGoal* or *ElseGoal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *ThenGoal* or *ElseGoal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *ThenGoal* or *ElseGoal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate differs from `->/2` by delaying the execution of *ThenGoal* or *ElseGoal* until the satisfiability of *Condition* has been determined. The other difference is that `domain_if/2` succeeds, if *Condition* is not satisfied.

The predicates `domain_if/2/3` are metapredicates and call their goals in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- domain_if( (X?\= 3),             % Condition
| (write('X not equal 3'), nl),            % ThenGoal
| (write('X equal 3'), nl)),               % ElseGoal
| write('Test: '), nl, (X = 2; X = 3).
Test:
X not equal 3

X       = 2 ;
X equal 3

X       = 3 ;

no
```

See also

?=/2, ?\=/2, ?</2, ?=</2, ?>/2, ?>=/2

Query domain limits

domain_maximum(+*Variable*, ?*Maximum***)**
domain_minimum(+*Variable*, ?*Minimum***)**

The predicate `domain_maximum/2` unifies *Maximum* with the largest value in the domain for *Variable*. The predicate `domain_minimum/2` unifies *Minimum* with the smallest value in the domain for *Variable*. If *Variable* is a domain integer, *Maximum* resp. *Minimum* is unified with it.

Arguments

Variable	Domain variable
Maximum	Domain integer
Minimum	Domain integer

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- X in 13..19, X ?> 15,
| domain_minimum(X, Minimum), domain_maximum(X, Maximum).

X       = __1
Minimum = 16
Maximum = 19

yes
```

See also

domain/2, lmaxdomain/2, lmindomain/2

Query domain size

domain_size(+*Variable*, ?*Cardinality*)

The predicate `domain_size/2` unifies *Cardinality* with the number of values in the domain of *Variable*. If *Variable* is a domain integer, *Cardinality* is unified with 1.

Arguments

Variable	Domain variable
Cardinality	Integer

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- X in [3, 5, 7, 8, 10], domain_size(X, Card).

X       = __1
Card    = 5

yes

[user] ?- X in [3, 5, 7, 8, 9], domain_size(X, L1),
| X ?\= 7, domain_size(X, L2),
| X = 9, domain_size(X, L3).

X       = 9
L1      = 5
L2      = 4
L3      = 1

yes
```

See also

domain/2

Generate constraint for list elements

element(?*Index*, @*List*, ?*Value*)

The predicate `element/3` generates a constraint between the positions and the values of list elements. The constraint is satisfied when *Value* is equal to the element at position *Index* in the *List*. The first list element has the *Index* 1.

If *Index* is less than 1 or greater than the number of list elements, the constraint cannot be satisfied. If *Index* is an integer, *Value* is unified with the list element at position *Index*. If *Value* is an integer, *Index* is restricted to those list positions at which the list element is equal to *Value*.

Arguments

Index	Domain variable
List	List of domain integers
Value	Domain variable

Exceptions

instantiation_error
The argument *List* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_integer)
An element of *List* must be a domain integer, but is a term of another type.

Example

```
[user] ?- element(X, [2,4,2,5], Y), Y ?< 4, X ?> 1.

X       = 3
Y       = 2

yes

[user] ?- element(X, [2,4,2,5], Y), Y = 5.

X       = 4
Y       = 5
```

```
yes

[user] ?- element(X, [2,4,2,5], Y),X ?> 2, Y ?> 3.

X       = 4
Y       = 5

yes

[user] ?- element(X, [3,1,4,5,2], C), X ?=< 2, domain(C, D).

X       = __1
C       = __2
D       = [1,3]

yes

[user] ?- element(X, [3,1,4,5,2], C), C ?>= 3, domain(X, D).

X       = __1
C       = __2
D       = [1,3,4]

yes

[user] ?- element(X, [3,1,4,5,2], C), X ?\= 3, domain(C, D).

X       = __1
C       = __2
D       = [1,2,3,5]

yes

[user] ?- element(X, [3,1,4,5,2], C), C ?\= 1, domain(X, D).

X       = __1
C       = __2
D       = [1,3,4,5]

yes
```

See also

domain/2

Generate number of identical values in a list

exactly(+*Number*, ?*List*, +*Value*)

The predicate `exactly/3` generates a constraint between two domain integers and the elements of a list. The constraint is satisfied when exactly *Number* list elements have the same *Value*.

If *List* is a variable, it is unified with a list which has exactly *Number* elements with the same *Value*.

Arguments

Number	Domain integer
List	List of domain variables
Value	Domain integer

Exceptions

instantiation_error
The argument *Number* or *Value* must not be a variable, but a variable was specified.

type_error(domain_integer)
The argument *Number* or *Value* must be a domain integer, but is a term of another type.

Example

You can implement `exactly/3` yourself using the predicate `cardinality/3`:

```
[user] ?- [user].
> my_exactly(Number, List, Value) :-
|       collect(List, Value, Conditions),
|       cardinality(Number, Number, Conditions), !.
> collect([], _, []).
> collect([V|R], Value, [V ?= Value|Conditions]) :-
|       collect(R, Value, Conditions).
> end_of_file.
*** consult 'user': loaded in 0.01 sec.

yes

[user] ?- exactly(4, L, 15).
```

```
L       = [15,15,15,15]

yes
```

See also

all_distinct/1, atmost/3, cardinality/3

Delimit domain

?*Variable* **in** @*Domain*
?*VariableList* **in** @*Domain*

The predicate `in/2` generates a constraint between a *Variable* or elements of a *VariableList* and the specified domain. The constraint is satisfied when all arguments lie within the specified domain. If *Variable* or an element of *VariableList* is uninstantiated, it is limited to the specified domain. If *Variable* or an element of *VariableList* is instantiated, the constraint is satisfied if the value of the variable lies within the specified domain.

The finite domain can be specified in three different ways:

- as an **interval** (*Min*:*Max*)
- as an **enumeration** of integers (sorted list *List*)
- as a **sequence** of integers (*Min*..*Max*)

If a *Variable* already had a domain, the intersection resulting from the old and the specified domains becomes the new domain. The resulting intersection is always at least as specific as the old and the specified domain:

- An intersection between an interval and an enumeration is an enumeration
- An intersection between an interval and a sequence is a sequence
- An intersection between an enumeration and a sequence is a sequence

If the intersection consists of only one value, *Variable* or each element of *VariableList* is unified with it. If no value remains, the predicate `in/2` fails.

The atom `in` is defined as an infix operator with precedence 750 and associativity `xfx`.

Arguments

Variable	Term
VariableList	List of terms
Domain	Finite domain
Min	Domain integer
Max	Domain integer
List	Sorted list of domain integers

Exceptions

instantiation_error
The argument *Domain* or a subterm must not be a variable, but a variable was specified.

type_error(domain)
The argument *Domain* must be a permitted domain. but is a term of an another type.

type_error(domain_integer)
The argument *Min* or *Max* must be a domain integer, but is a term of another type.

type_error(domain_integer)
An element of *List* must be a domain integer, but is a term of another type.

domain_error(sorted_list)
The argument *List* is not sorted in ascending order.

Example

```
[user] ?- A in -10..5, B in -3..6,
| C in [-16, -6, -2, 0, 1, 4, 8],
| domain(A, DA), domain(B, DB), domain(C, DC).

A       = __1
B       = __2
C       = __3
DA      = [-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5]
DB      = [-3,-2,-1,0,1,2,3,4,5,6]
DC      = [-16,-6,-2,0,1,4,8]

yes

[user] ?- A in -10:5, A in -3..6,
| A in [-16, -6, -2, 0, 1, 4, 8],
| domain(A, DA).

A       = __1
DA      = [-2,0,1,4]

yes

[user] ?- A in 3..8, A in 6:10, A in [3, 5, 7].

A       = 7
```

```
yes

[user] ?- Min = 4, Max = 20, [6, 9, 17] in Min:Max.

Min     = 4
Max     = 20

yes
```

See also

domain/2, is_in_domain/2

Generate value

# indomain(?*Variable*)

# indomain(?*Variable*, +*Heuristic*)

The predicates `indomain/1/2` generate for *Variable* a value which satisfies the currently applicable constraints.

The argument *Heuristic* specifies the order in which values are generated by backtracking:

`minimum`	The values are assigned in ascending order
`maximum`	The values are assigned in descending order
`center`	A middle value is generated first, and the subsequent values are alternately greater and smaller than the first
`random`	A random value is generated first, and the subsequent values are alternately greater and smaller than the first
domain integer *Value*	A value smaller than or equal to *Value* is generated first, and the subsequent values are alternately greater and smaller than the first

If *Variable* is a domain variable, consistent instantiations are generated in given order by means of backtracking.
If *Variable* is a domain integer, the predicate succeeds deterministically, if *Heuristic* is an atom or if *Heuristic* is a domain integer *Value* equal to *Variable*.

The predicate `isdomain/1` is equivalent to `isdomain/2` with *Heuristic* = `minimum`.

Arguments

Variable	Domain variable
Heuristic	Atom: `minimum | maximum | center | random`
	Domain integer

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

instantiation_error
The argument *Heuristic* or a subterm must not be a variable, but a variable was specified.

domain_error(heuristic)
An element of *Heuristic* is not within the range for heuristics.

Example

```
[user] ?- X in [4, 8, 11], X ?\= 8, indomain(X).

X       = 4 ;

X       = 11 ;

no

[user] ?- X in [4, 8, 11], X ?\= 8, indomain(X, maximum).

X       = 11 ;

X       = 4 ;

no

[user] ?- X in 5..9, X ?\= 8, indomain(X, center).

X       = 7 ;

X       = 6 ;

X       = 9 ;

X       = 5 ;

no
```

Compatibility

V5.1A The predicate `indomain/2` is new.

See also

label/1/2

Test variable type

is_consecutive(+*Term*)

The predicate `is_consecutive/1` succeeds if *Term* is a domain variable of the sequence type; otherwise, it fails.

Arguments

Term	Term

Example

```
[user] ?- V in 3..7, is_consecutive(V).

V       = __1

yes

[user] ?- V in 3:7, is_consecutive(V).

no

[user] ?- V in [3, 6, 8], V in 2..9, is_consecutive(V).

V       = __1

yes
```

See also

in/2, is_domain/1, is_interval/1, is_enumeration/1

Test variable type

is_domain(+*Term*)

The predicate `is_domain/1` succeeds if *Term* is a domain variable; otherwise, it fails.

Arguments

Term	Term

Example

```
[user] ?- V in 3..7, is_domain(V).

V       = __1

yes

[user] ?- is_domain(W).

no
```

See also

in/2, is_consecutive/1, is_interval/1, is_enumeration/1

Test variable type

is_enumeration(+*Term*)

The predicate `is_enumeration/1` succeeds if *Term* is a domain variable of the enumeration type; otherwise, it fails.

Arguments

Term	Term

Example

```
[user] ?- V in [3, 6, 8], is_enumeration(V).

V       = __1

yes

[user] ?- V in 3..7, is_enumeration(V).

no

[user] ?- V in [3, 6, 8], V in 2..9, is_enumeration(V).

no
```

See also

in/2, is_consecutive/1, is_domain/1, is_interval/1

Test domain affiliation

is_in_domain(+*Variable*, +*Value*)

The predicate `is_in_domain/2` checks whether *Value* lies within the domain of *Variable*.

The predicate succeeds if *Variable* is a domain variable and *Value* lies within its domain, or if *Variable* is a domain integer equal to *Value*.

Arguments

Variable	Domain variable
Value	Domain integer

Exceptions

instantiation_error
The argument *Value* must not be a variable, but a variable was specified.

type_error(domain_integer)
The argument *Value* must be a domain integer, but is a term of another type.

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- X in [1, 3, 5], is_in_domain(X, 5).

X       = __1

yes

[user] ?- X in [1, 3, 5], X ?= 5.

X       = 5

yes
```

See also

domain/2, ?=/2, indomain/2

Test variable type

is_interval(+*Term*)

The predicate `is_interval/1` succeeds if *Term* is a domain variable of the interval type; otherwise, it fails.

Arguments

Term	Term

Example

```
[user] ?- V in 3:7, is_interval(V).

V       = __1

yes

[user] ?- V in 3..7, is_interval(V).

no
```

See also

in/2, is_consecutive/1, is_domain/1, is_enumeration/1

Generate values

# label(?*List*)

The predicate `label/1` generates consistent instantiations for the domain variables in the *List* in ascending order by means of backtracking.

Arguments

List	List of domain variables

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

Example

You can define the predicate `label/1` yourself:

```
[user] ?- [user].
> my_label([]).
> my_label([V|R]) :- indomain(V), my_label(R).
> end_of_file.
*** consult 'user': loaded in 0.01 sec.

[user] ?- L = [A, B, C], A in 3..5, B = 2, C in [8, 12],
| label(L),
| write(L), nl, fail.

[3,2,8]
[3,2,12]
[4,2,8]
```

```
[4,2,12]
[5,2,8]
[5,2,12]

no
```

See also

indomain/1/2, label/2, label_tb/3, deleteff/3, deleteff0/3, deleteffc/3

Generate values

# label(?*List*, +*Heuristic*)

The predicate `label/2` generates consistent instantiations for the domain variables in the *List* by means of backtracking.

Heuristic is a list with at most two elements. It allows you to determine the order in which variables are instantiated and how values are generated.

The following atoms determine, in which order the variables in *List* are chosen:

`card`	The variable with the smallest domain is to be instantiated.
`constraint`	The variable with the most active constraints is to be instantiated.

The following atoms determine, in which order the values of a domain are chosen:

`maximum`	The values are to be generated in descending order.
`minimum`	The values are to be generated in ascending order.
`center`	A center value is to be generated first, and the subsequent values are to be alternately greater and smaller than the first.
`random`	A random value is to be generated first, and the subsequent values are to be alternately greater and smaller than the first.

Arguments

List	List of domain variables
Heuristic	List of atoms

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

instantiation_error
The argument *Heuristic* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* or *Heuristic* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(atom)
An element of *Heuristic* must be an atom, but is a term of another type.

domain_error(heuristic)
An element of *Heuristic* is not within the range for heuristics.

Example

```
[user] ?- L = [A, B, C],
| A in 3..5, B = 2, C in [5, 12], A ?\= C,
| label(L, [maximum, card]),
| write(L), nl, fail.

[5,2,12]
[4,2,12]
[3,2,12]
[4,2,5]
[3,2,5]

no
```

See also

indomain/1/2, label/1, label_tb/3, deleteff/3, deleteff0/3, deleteffc/3

Generate values

# label_tb(?*List*, +*Time*, +*Heuristic*)

The predicate `label_tb/3` generates consistent instantiations for the domain variables in the *List* by means of backtracking.

Time specifies a time limit in CPU seconds. If this time limit is exceeded, the search is aborted and the predicate fails.

Please refer to predicate `label/2` for the description of the argument *Heuristic*.

Arguments

List	List of domain variables
Time	Number or evaluable expression
Heuristic	List of atoms

Exceptions

instantiation_error
The argument *List* or *Time* must not be a variable, but a variable was specified.

instantiation_error
The argument *Heuristic* or a subterm must not be a variable, but a variable was specified.

type_error(list)
The argument *List* or *Heuristic* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(atom)
An element of *Heuristic* must be an atom, but is a term of another type.

domain_error(heuristic)
An element of *Heuristic* is not within the range for heuristics.

type_error(number)
The argument *Time* must be a number or an arithmetic expression, but is a term of another type.

Furthermore, if an arithmetic expression is specified for the argument *Time*, all the exceptions for `is/2` can occur.

Example

```
[user] ?- A in [3, 5, 7], B in 8..10,
| label_tb([A, B], 1, [minimum]),
| write([A, B]), nl, fail.

[3,8]
[3,9]
[3,10]
[5,8]
[5,9]
[5,10]
[7,8]
[7,9]
[7,10]

no

[user] ?- A in [3, 5, 7], B in 8..10,
| label_tb([A, B], 1e-8, [minimum]),
| write([A, B]), nl, fail.
[3,8]
[3,9]
[3,10]
[5,8]
[5,9]
[5,10]

no
```

See also

indomain/1/2, label/1/2, deleteff/3, deleteff0/3, deleteffc/3

Query domain limits

lmaxdomain(@*List*, ?*Maximum*)
lmindomain(@*List*, ?*Minimum*)

The predicate `lmaxdomain/2` unifies *Maximum* with the greatest maximum of domains for the variables in *List*.

The predicate `lmindomain/2` unifies *Minimum* with the smallest minimum of domains for the variables in *List*.

If *List* is an empty list, the predicates fail.

Arguments

List	List of domain variables
Minimum	Domain integer
Maximum	Domain integer

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- A in [3, 5, 7], B in 8..10, C in -2..3, D in 0:3,
| L = [A, B, C, D],
| lmaxdomain(L, Max),
| lmindomain(L, Min).

A       = __1
B       = __2
C       = __3
```

```
D       = __4
L       = [__1,__2,__3,__4]
Max     = 10
Min     = -2

yes
```

See also

domain_maximum/2, domain_minimum/2, lmaxmin/2, lminmax/2

Query domain limits

lmaxmin(@*List*, ?*MaxMin*)
lminmax(@*List*, ?*MinMax*)

The predicate `lmaxmin/2` unifies *MaxMin* with the greatest minimum of domains for the variables in *List*.

The predicate `lminmax/2` unifies *MinMax* with the smallest maximum of domains for the variables in *List*.

If *List* is an empty list, the predicates fail.

Arguments

List	List of domain variables
MaxMin	Domain integer
MinMax	Domain integer

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

Example

```
[user] ?- A in [3, 5, 7], B in 8..10, C in -2..3, D in 0:3,
| L = [A, B, C, D],
| lminmax(L, MinMax),
| lmaxmin(L, MaxMin).

A       = __1
B       = __2
C       = __3
```

```
D       = __4
L       = [__1,__2,__3,__4]
MinMax  = 3
MaxMin  = 8

yes
```

See also

domain_maximum/2, domain_minimum/2, lmaxdomain/2, lmindomain/2

Constraint for maximum and minimum values of a list

maximum(+*List*, ?*Maximum*)
minimum(+*List*, ?*Minimum*)

The predicate `maximum/2` generates a constraint where *Maximum* lies between the greatest minimum and the greatest maximum of the domains of the list elements.

The predicate `minimum/2` generates a constraint where *Minimum* lies between the smallest minimum and the smallest maximum of the domains of the list elements.

If an element in the *List* is neither a domain variable nor an integer, the predicates fail.

Arguments

List	List of domain variables
Maximum	Domain variable
Minimum	Domain variable

Exceptions

instantiation_error
The argument *List* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

Example

```
[user] ?- X ?>= 0, Y ?< 0, maximum([X, Y], Z).

X       = __3
Y       = __2
Z       = __3

yes

[user] ?- A in 1:3, B in 5:7, maximum([A, B], 6).

A       = __1
B       = 6

yes
```

```
[user] ?- A in 4..8, B in 10..15,
| maximum([A, B], C), domain(C, DC).

A       = __1
B       = __3
C       = __3
DC      = [10,11,12,13,14,15]

yes

[user] ?- [X0, Y0] in 1:3, [X1, Y1] in 2:4,
| minimum([X0, X1], X), maximum([Y0, Y1], Y),
| Y ?< X.

X0      = 3
Y0      = __2
X1      = __3
Y1      = 2
X       = 3
Y       = 2

yes

[user] ?- A in 4..8, B in 2..6, C in [-2, 2, 6, 10],
| maximum([A, B, C], Max), domain(Max, DMax),
| minimum([A, B, C], Min), domain(Min, DMin).

A       = __1
B       = __2
C       = __3
Max     = __5
DMax    = [4,5,6,7,8,9,10]
Min     = __7
DMin    = [-2,-1,0,1,2,3,4,5,6]

yes
```

See also

domain_maximum/2, domain_minimum/2, lmaxdomain/2, lmindomain/2

Optimize result value

minimize_bb(+*Goal*, ?*Optimum*) `[ @ +`*Module* `]`
minimize_bb(+*Goal*, ?*Optimum*, +*Lower*, +*Upper*, +*Percent*) `[ @ +`*Module* `]`

The predicates `minimize_bb/2/5` search for a minimum value for *Optimum*. A successful proof of *Goal* must instantiate *Optimum* with an integer. Before calling the predicate, at least the maximum of *Optimum* must be appropriately constrained. IF/Prolog attempts to prove *Goal* using the **branch-and-bound** method in such a way as to further minimize the value of *Optimum*. With this method, an attempt is made to find an alternative solution from the point at which the lower limit for *Optimum* was most recently incremented.

The predicate `minimize_bb/5` constrains the range of *Optimum* between *Lower* and *Upper*. *Percent* is used to define the maximum percentage by which the minimum found may exceed the actual minimum. This is achieved by constraining the next solution to be at least *Percent* % better than the current optimum.

Arguments

Goal	Goal
Optimum	Domain variable
Lower	Domain integer
Upper	Domain integer
Percent	Integer between 0 and 100

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

instantiation_error
: The argument *Lower*, *Upper* or *Percent* must not be a variable, but a variable was specified.

type_error(domain_variable)
: The argument *Optimum* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
: The argument *Optimum* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(domain_integer)
: The argument *Lower* or *Upper* must be a domain integer, but is a term of another type.

type_error(integer)
: The argument *Percent* must be an integer, but is a term of another type.

domain_error(percentage)
: The argument *Percent* is not in the range 0..100.

Hints

The predicates `minimize_bb/2/5` are metapredicates and call their goals in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- X in 20:50,
| minimize_bb(indomain(X, maximum), X).

X       = 20

yes
```

See also

minimize_maximum/2/5, label/1/2, label_tb/3, indomain/1/2

Optimize

minimize_maximum(+*Goal*, ?*List*) [@ +*Module*]
minimize_maximum(+*Goal*, ?*List*, +*Lower*, +*Upper*, +*Percent*) [@ +*Module*]

The predicates `minimize_maximum/2/5` minimize the maximum value of the elements of *List* which satisfies *Goal*. A successful proof of *Goal* must instantiate the elements of *List* with integers. IF/Prolog attempts to prove the *Goal* using the **branch-and-bound** method in such a way as to further minimize the maximum value for the list elements. Once one solution has been found, the *Goal* is reactivated with the additional constraint that the next solution must be better than the previous one. This process is repeated until no better solution can be found.

The predicate `minimize_maximum/5` constrains the range of the list elements between *Lower* and *Upper*. *Percent* is used to define the maximum percentage by which the minimum found may exceed the actual minimum. This is achieved by constraining the next solution to be at least *Percent* % better than the current optimum.

Arguments

Goal	Goal
List	List of domain variables
Lower	Domain integer
Upper	Domain integer
Percent	Integer between 0 and 100

Exceptions

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

instantiation_error
The argument *List*, *Lower*, *Upper* or *Percent* must not be a variable, but a variable was specified.

type_error(list)
The argument *List* must be a regular list, but is a term of another type or not regular.

type_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
An element of *List* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(domain_integer)
The argument *Lower* or *Upper* must be a domain integer, but is a term of another type.

type_error(integer)
The argument *Percent* must be an integer, but is a term of another type.

domain_error(percentage)
The argument *Percent* is not in the range 0..100.

Hints

The predicates `minimize_maximum/2/5` are metapredicates and call their goals in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- [X, Y, Z] in 0:10, X + Y ?= 10, X + Z ?= 12,
| minimize_maximum(label([X, Y, Z]), [X, Y, Z]).

X       = 6
Y       = 4
Z       = 6
```

```
yes

[user] ?- X in 20:100,
| minimize_maximum(indomain(X, maximum), [X]).

X       = 20

yes

[user] ?- X in 20:100,
| minimize_maximum(indomain(X, maximum), [X], 20, 60, 33).

X       = 24

yes
```

See also

minimize_bb/2/5, label/1/2, label_tb/3, indomain/1/2

Monitor variable domain

monitor_domain(+*Variable*, +*Modification*, ?*Goal*) `[ @ +`*Module* `]`

The predicate `monitor_domain/3` attaches a goal to the domain variable *Variable* to be called when the domain of the variable changes.

The argument *Modification* specifies the art of the modification:

`bind`	The variable has been instantiated
`boundary`	A boundary (lower or upper limit) of the variable domain has been changed
`modify`	The variable domain has been changed

When the specified modification takes place, the goal *Goal* is implicitly called.

Arguments

Variable	Domain variable
Modification	Atom: bind | boundary | modify
Goal	Goal

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

instantiation_error
The argument *Modification* must not be a variable, but a variable was specified.

instantiation_error
The argument *Goal* must not be a variable, but a variable was specified.

type_error(callable)
The argument *Goal* must have the syntactical structure of a Prolog goal.

existence_error(procedure)
In executing *Goal*, a predicate was to be activated which is not defined and the Prolog flag `unknown` has the value `error`.

type_error(atom)
The argument *Goal* or a subgoal has been qualified by means of `@/2` or `:/2` with a term that is not an atom.

existence_error(module)
The argument *Goal* or a subgoal is qualified by means of `@/2` or `:/2` with an atom that does not name an existing module.

Hints

The predicate `monitor_domain/3` is a metapredicate and calls its goal in the calling module or in the specified *Module*.

The predicates activated in the goal must be visible in the calling module or in the specified *Module*, unless the `:/2` qualification is used for such a predicate to indicate explicitly the module in which this predicate is visible.

The predicates activated in the goal are normally executed in the context of the module in which they are defined. This does not apply to metapredicates, which are executed in the context of calling module or the specified *Module*, unless the `@/2` qualification is used for a metapredicate to indicate explicitly the module context in which this predicate is to be executed.

Example

```
[user] ?- A in 1:10,
        monitor_domain(A, boundary, domain(A,DA)),
        A ?> 3.

A       = __1
DA      = [4,5,6,7,8,9,10]

yes

[user] ?- A in [1,3,5,7],
        monitor_domain(A, modify, domain(A,DA)),
        A ?\= 3.

A       = __1
DA      = [1,5,7]

yes

[user] ?- A in 10..20,
        monitor_domain(A, bind, domain(A,DA)),
        A ?> 19.

A       = 20
DA      = [20]

yes
```

Compatibility

V5.1A The predicate `monitor_domain/3` is new.

See also

domain/2

Remove interval from domain

notin(+*Variable*, +*From*, +*To*)

The predicate `notin/3` removes all values between *From* and *To* from the domain of *Variable*. If the domain becomes empty, the predicate fails. If the domain contains only one value, *Variable* is instantiated to this value.

The satisfiability of the constraint can be immediately decided for variables of **enumeration** and **sequence** type. For variables of **interval** type, satisfiability can be decided either immediately or later, when more information becomes available.

Arguments

Variable	Domain variable
From	Domain integer
To	Domain integer

Exceptions

type_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a term of another type.

domain_error(domain_variable)
The argument *Variable* must be a domain variable or a domain integer, but is a **normal** variable.

type_error(domain_integer)
The argument *From* or *To* must be a domain integer, but is a term of another type.

Example

```
[user] ?- X in [3, 5, 7, 9], notin(X, 6, 10), domain(X, L1).

X       = __1
L1      = [3,5]

yes

[user] ?- X in 1..10, notin(X, 3, 5), domain(X, L1).

X       = __1
L1      = [1,2,6,7,8,9,10]
```

```
yes

[user] ?- A in 1:10, notin(A,1,4), notin(A,6,20).

A       = 5

yes
```

Compatibility

V5.1A The predicate `notin/3` is new.

See also

in/2, ?\=/2, indomain/1/2

Establish relations

relation(?*VariableList*, +*Table*)

The predicate `relation/2` defines multiple relations over finite sets by enumeration of tuples.

Each relation is specified as a list of domain integers in *Table*. The elements of *VariableList* are restricted to the value combinations specified in the tuple list *Table*. The list *VariableList* and each tuple in *Table* must have the same number of elements.

Arguments

VariableList	List of domain variables
Table	List of lists of domain integers

Exceptions

instantiation_error
The argument *VariableList*, *Table* or an element of *Table* must not be a variable, but a variable was specified.

type_error(list)
The argument *VariableList*, *Table* or an element of *Table* must be a regular list, but is a term of another type or not regular.

type_error(domain_integer)
An element of *Table* must be a domain integer, but is a term of another type.

Hints

The predicate can be used to describe relations that are difficult or impossible to define arithmetically.

Example

```
[user] ?- relation([X1, X2, X3], [[0, 0, 1], [1, 1, 0]]),
| (X1 = 0; X3 = 0; (X1 = 0, X3 = 0)).

X1      = 0
X2      = 0
X3      = 1 ;

X1      = 1
X2      = 1
```

```
X3      = 0 ;

no

% Explicit representation of relations
%       [Q,R,X,Y] in [0,1,2,3]
%       X ?= Q*Y+R
%       0 ?=< R ?< Y
?- relation([Q,R,X,Y],
|       [
|       [0,0,0,1], [0,0,0,2], [0,0,0,3], [0,1,1,2],
|       [0,1,1,3], [0,2,2,3], [1,0,1,1], [1,0,2,2],
|       [1,0,3,3], [1,1,3,2], [2,0,2,1], [3,0,3,1]
|       ]).
```

The compatibility of devices, which are represented by numbers, can be expressed using the following predicate:

```
compatible(Device1, Device2) :-
        relation([Device1, Device2],
                [
                        [135500, 135501],
                        [135500, 135502],
                        [135500, 135503],
                        [135502, 135503],
                        [135501, 135500]
                ]).
```

Compatibility

V5.1A The list length of the arguments is no more limited to 127.

Chapter 5

Boolean constraints

Boolean constraints are conditions for a variable with the domain 0 and 1. The Boolean domain is thus a small subdomain of the finite domain. Consequently Boolean constraints are defined as a subclass of finite constraints.

The numbers correspond to the truth values: 1 corresponds to the value `TRUE`, 0 to the value `FALSE`. Boolean constraints are formulated by combining two Boolean expressions with a Boolean operator.

If you wish to work with Boolean constraints, you must import the module `const_domain` in which built-in predicates for Boolean constraints are stored in addition to those for finite constraints.

Use the `import` directive to import the module into your database:

```
[user] ?- [user].↩
> :- import(const_domain).↩
> end_of_file.↩
*** consult 'user': loaded in 0.02 sec.

yes
```

i You can only work with constraints only if you have configured the rational number support and the constraint package at installation.

5.1 Syntax of Boolean constraints

The operands of Boolean constraints are Boolean expressions. A Boolean expression is either a variable in the range 0 through 1 or any Boolean operation using these variables. The syntax of a Boolean constraint looks like this:

$$BoolConstraint ::= \ BoolExpression \quad BoolOperator \quad BoolExpression$$

$$BoolExpression ::= \left\{ \begin{array}{lll} Variable & & \\ 0 \mid 1 & & \\ BoolExpression & \backslash/ & BoolExpression \\ BoolExpression & /\backslash & BoolExpression \\ BoolExpression & \# & BoolExpression \\ BoolExpression & => & BoolExpression \\ BoolExpression & <=> & BoolExpression \\ \tilde{}BoolExpression & & \\ (BoolExpression) & & \end{array} \right\}$$

$$BoolOperator ::= \left\{ \begin{array}{c} => \\ <=> \end{array} \right\}$$

A Boolean constraint is thus a combination of two Boolean expressions. Each Boolean expression may contain any number of variables.

5.2 Built-in operators and predicates

This section provides a tabular overview of the built-in predicates followed by descriptions of the predicates in alphabetical order.

The following built-in predicates are assigned to the module `const_domain`; consequently you can call them only if you have imported this module.

The built-in operators have been extended to include the Boolean operators which allow you to combine two Boolean expressions in a constraint. The operators generate a constraint which is satisfiable when the Boolean expressions on both sides satisfy the specified relation.

Operator	**Precedence**	**Type**	**Meaning**
\/	500	yfx	Logical OR (disjunction)
/\	500	yfx	Logical AND (conjunction)
#	500	xfx	Logical XOR (antivalence)
=>	800	xfx	Implication
<=>	800	xfx	Equivalence
~	400	xfx	Negation

Boolean equivalence

$\boxed{\text{?}\textit{BoolExpression1} <=> \textit{?BoolExpression2}}$

The predicate <=>/2 generates a constraint between Boolean expressions. The constraint is satisfied when both expressions have the same truth value. The constraint is based on the following Boolean axioms:

$$\texttt{true} \leftrightarrow \texttt{true}$$
$$\texttt{false} \leftrightarrow \texttt{false}$$

The atom <=> is defined as an infix operator with precedence 800 and associativity `xfx`.

Arguments

BoolExpression1	Boolean expression
BoolExpression2	Boolean expression

Exceptions

type_error(boolean)
The argument *BoolExpression1* or *BoolExpression2* must be a Boolean expression, but is a term of another type.

Example

```
[user] ?- A /\ B <=> 1.

A       = 1
B       = 1

yes

[user] ?- A /\ B <=> A \/ B, indomain(A).

A       = 0
B       = 0 ;

A       = 1
B       = 1 ;

no

[user] ?- ((( X => Y) => Z) <=> (X => (Y => Z))) <=> 0,
                label([X,Y,Z]).

    X       = 0
    Y       = 0
    Z       = 0 ;

    X       = 0
    Y       = 1
    Z       = 0 ;

    no
```

See also

=>/2

Boolean implication

?*BoolExpression1* => ?*BoolExpression2*

The predicate `=>/2` generates a constraint between Boolean expressions. The constraint is satisfied if the second expression is implied by the first expression, i.e. if *BoolExpression1* is false or if *BoolExpression2* is true. The constraint is based on the following Boolean axioms:

$$\texttt{true} \rightarrow \texttt{true}$$
$$\texttt{false} \rightarrow *$$

The atom => is defined as an infix operator with precedence 800 and associativity `xfx`.

Arguments

BoolExpression1	Boolean expression
BoolExpression2	Boolean expression

Exceptions

type_error(boolean)
The argument *BoolExpression1* or *BoolExpression2* must be a Boolean expression, but is a term of another type.

Example

```
% A simple puzzle: visit from family P.
% Mr. and Mrs. P. have three children: Julia, Sonja and Anita.
% 1. If Mr. P comes, then he'll bring his wife with him.
% 2. At least one of the daughters Sonja and Anita will come.
% 3. Either both Mrs. P and Julia come, but not both of them.
% 4. Either both Julia and Sonja come, or neither of them.
% 5. If Anita comes, then Sonja and Mr. P will also come.

[user] ?- Mr_P_comes => Mrs_P_comes,
          Anita_comes \/ Sonja_comes <=> 1,
          Mrs_P_comes # Julia_comes <=> 1,
          Julia_comes <=> Sonja_comes,
          Anita_comes => Sonja_comes /\ Mr_P_comes,

          label([ Mr_P_comes, Mrs_P_comes, Julia_comes,
                  Sonja_comes, Anita_comes]).

Mr_P_comes      = 0
```

```
Mrs_P_comes     = 0
Anita_comes     = 0
Sonja_comes     = 1
Julia_comes     = 1

yes
```

See also

<=>/2

Bibliography

[1] IF/Prolog V5.3 Reference Manual

[2] IF/Prolog V5.3 User's Guide

[3] IF/Prolog V5.3 OSF/Motif Interface

[4] IF/Prolog V5.3 Informix Interface

[5] IF/Prolog V5.3 Constraints Package

[6] IF/Prolog V5.3 Quick Reference

[7] IF/Prolog V5.3 Windows Interfaces

[8] IF/Prolog V5.3 Java Interface

[9] IF/Prolog V5.3 BDD Package

[10] X/Open CAE (Common Applications Environment) Specification. System Interfaces and Headers, Issue 4. Prentice Hall 1994.

[11] International Standard, ISO/IEC IS 13211-1. International Organization for Standardization, 1995

[12] William F. Clocksin, Chris S. Mellish: Programming in PROLOG. Standard Edition Berlin et al.: Springer 1995.

[13] Ivan Bratko: PROLOG. Programming for Artificial Intelligence Second Edition, Addison-Wesley 1990.

[14] Leon Sterling, Ehud Shapiro: The Art of PROLOG. Advanced Programming Techniques. Cambridge Massachusetts: MIT Press, 1986

[15] Pascal van Hentenryck: Constraint Satisfaction in Logic Programming Cambridge Massachusetts: MIT Press, 1989

Index