CS-Prolog II

Version 2.3

User's Manual

ML Consulting and Computing Ltd.

Budapest, Hungary

January, 1999

8 January 1999

The information in this document is subject to change without notice and should not be construed as a commitment by ML Consulting and Computing Ltd. ML Consulting and Computing Ltd. does not assume any responsibility for any errors that may appear in this document.

There is no post-paid reader's comment form at the end of this document. Please address your comments (with the product version data included) directly to ML Ltd., H-1011 Budapest, Gyorskocsi u. 5-7; or send by e-mail to **mail@ml-cons.hu**.

Contents

Overview

CS-Prolog II is a Prolog system developed in Hungary during years 1996-1999. The base for defining the syntax and built-in predicates of CS-Prolog was the Prolog Standard (ISO/IEC 13211-1) which appeared in 1995. This base was extended with several features not included in the present standard — modularity, multi-processing, real-time programming.

Beginning with version 2.0 a networking communication facility, an external data base access facility, and a WWW interface had been added to the system. These extensions were funded by the EU in the framework of the INCO-Copernicus project *ExperNet: A Distributed Expert System for the Management of a National Network*, No 960114.

The most interesting property of CS-Prolog II is the possibility of creating processes that run in parallel. It gives the opportunity of implementing parallel algorithms. CS-Prolog runs on different architectures having either one or more processors. Processes created on the same processor are executed in quasi-parallel. Processes can communicate with each other through channels. Real-time techniques are available as, e.g., creation of cyclic processes, alarm clock to set time-outs, etc. The networking communication is an inter-application extension of the channel concept.

A sophisticated exception handling mechanism is implemented in CS-Prolog II that makes it possible to catch an exceptional situation (an error or an interrupt) and handle it either on the place of the occurrence or on a higher level, and to continue the execution afterwards.

The CS-Prolog II programming system consists of a compiler, a linker, a runtime system and an integrated programming environment that combines the previous three components. In the environment, the user can debug the program using an interactive trace. The compiler incorporates a preprocessor, which is very similar to a standard C preprocessor. It helps to write more readable and portable programs.

This user manual describes the CS-Prolog II programming language and the programming system. This book cannot serve as a primer to learn Prolog, it assumes that the reader is familiar with the Prolog language. The name of the product will be abbreviated as CS-Prolog or CSP-II, but in this text these abbreviations will mean always the CS-Prolog II system.

This volume is divided to four parts. These are:

I. CS-Prolog Language
II. Parallel and Real-Time Features
III. Built-in Predicates
IV. The Programming Environment

Part I describes the basic components of the language except the multitasking notions, which are collected in part II. The third part contains the description of the built-in predicates, and the fourth one describes the usage of the components of the programming system.

The networking features, the data base access, and the WWW interface, are described in separate supplements.

There used to be a second volume, now discontinued, for machine and operating system specific information. The latest releases of CSP-II run only on unix platforms; the installation kit contains all the necessary information.

Beginning with version 2.3 a Constraint Logical Programming (CLP) extension facility is included in the system optionally. The runtime component can be shipped in two versions: with or without the CLP extension. The version with CLP can have zero, one, or more (up to four) different CLP solvers installed. If no solver is installed, or the user program does not activate any solver explicitly, then the extended system behaves as the system without the extension (although the execution is somewhat slower).

An important characteristic of the CLP extension to be considered is that a new term type is introduced with it, and this changes the meaning of several built in predicates relying on the closed set of types defined in the Standard. Fortunately, this change has effect only when objects of this new type are actually present, i.e. when the user program explicitly activates a solver.

The general description of the CLP extension is contained in two separate chapters of this manual; occasional references are included in other parts, too. The description of particular solvers will be supplied as individual supplements.

There is an experimental linear solver called the ‘ML solver’ included in the distribution kit, that can be installed by the user. This solver is based on a linear programming algorithm; it handles linear inequalities and equations over real numbers. It has some known bugs, and at present it can be run only by one of the CS-Prolog processes at one time, so it is not recommended for use in production systems.

Note on version numbers: the major components of CS-Prolog II each have their own version number. The most frequently changing component is the runtime program so the version number on the title page corresponds to this number. Other components write out their own version number unless suppressed.

PART I

CS-Prolog Language

1. Syntax

CS-Prolog implements the standard Prolog syntax (it is close to the previous de facto standard Edinburgh Prolog syntax).

A CS-Prolog program consists of one or more separately compiled modules.

The CS-Prolog system contains a preprocessor, described in chapter 5. The modules usually are scanned and transformed by the preprocessor before the actual compilation. The following description deals with the syntax of a CS-Prolog module after preprocessing, when all preprocessor commands and macros had already been expanded.

1.1 Term syntax

A Prolog term is one of the following entities:

- constant term
- variable
- compound term

Constant terms are also called atomic terms.

1.1.1 Constant terms

A constant term can be:

- number
- atom

Numbers are unsigned integer or floating-point numbers. The prefix operator minus (-) with a numeric constant as its operand denotes the corresponding negative number.

External representation of integer numbers:

- integer literal
- character code literal
- binary literal
- octal literal
- hexadecimal literal

An integer literal is an unsigned decimal integer.

A character code literal begins with **0'** prefix (zero character, quote character) and is followed by a *single quoted character* (an item which can be part of a single quoted string, see section 1.1.11). It denotes the numerical value of the character.

A binary literal begins with **0b** prefix followed by one or more binary digits (0, 1). An octal literal begins with **0o** prefix followed by one or more octal digits (0 through 7). A hexadecimal literal begins with **0x** prefix followed by one or more hexadecimal digits (0 through 9 and either uppercase or lowercase **a** through **f**).

The external representation of a floating-point number is the usual floating point literal in scientific notation, the exponential part being optional (there must be a fractional part, however).

Examples:

`1994`
: means the decimal number 1994 (integer literal)

`0'a`
: means the decimal number 97 (character code literal)

`0'\'`
: means the decimal number 39 (character code literal - meta escape sequence)

`0'''`
: also means the decimal number 39 (character code literal)

`0b101`
: means the decimal number 5 (binary literal)

`0'\101\`
: means the decimal number 65 (character code literal - octal escape sequence)

`0o101`
: also means the decimal number 65 (octal literal)

`0xFF`
: means the decimal number 255 (hexadecimal literal)

`19.94`
: means the float number 19.94 (floating point literal)

`1.994E1`
: also means the float number 19.94 (floating point literal)

An atom can be:

- identifier token
- graphic token
- quoted token
- the empty list (**[]**)
- the empty curly brackets (**{}**)
- semicolon token (**;**)
- cut token (**!**)

An identifier is an arbitrary sequence of alphanumeric characters and underscore (**_**) character, which begins with a lowercase letter.

A graphic token is a character sequence composed from any of the following characters:

```
# $ & * + - . / : < = > ? @ ^ ~ \
```

A graphic token cannot begin with the character sequence of the comment open (**/***). A graphic token also cannot be the single character '.' (dot) when the dot is followed by whitespace or single line comment (it will be parsed as an end-of-term token in this case).

If a graphic token is followed by a comment open character sequence, they must be separated by at least one white-space character. E.g., the expression

```
@/* comment text */
```

is not treated by the compiler as a comment, because the character sequence **@/*** is a graphic token.

A quoted token is a character string constant, represented by a single quoted string, or a back quoted string, or in certain cases by a double quoted string (see section 1.1.11), i.e., a character sequence enclosed in a matching pair of single quote (**'**), double quote (**"**) or backquote (**`**) characters. A quoted token, which contains no character, is the null atom. A quoted string can be spread over two or more lines by means of continuation escape sequences. A continuation escape sequence is a backslash character (****) immediately followed by a newline. A quoted string containing one or more continuation escape sequences denotes the same item as the quoted string obtained by removing the continuation escape sequences from the original quoted string. If a quoted string is to contain one or more instances of the delimiting quote character, they are to be either doubled or escaped. In CSP-II, backquoted strings are treated like single quoted strings — they yield quoted tokens. The interpretation of double quoted strings is under control of the settable prolog option flag **double_quotes** (they can be interpreted as atoms, character lists, or character code lists).

Character code constants and quoted atoms may contain other escape sequences beside the continuation sequence. These are described in section 1.1.14.

The special atom

```
[]
```

is used by convention to denote the empty list.

The maximum length for an atom in CS-Prolog is limited in 4095 characters.

Examples of atoms:

```
apple_pie
applePie
'Apple pie'
<=+=>
''
[]
```

1.1.2 Variables

A variable is an arbitrary sequence of alphanumeric characters and the underscore character beginning with either an uppercase letter or the underscore character. A variable consisting of a single underscore character is called an *anonymous* variable. Each occurrence of the anonymous variable is different from any other occurrence of a non-anonymous or the anonymous variable.

Note: If a variable is only referred to once in a clause, it does not need to be named and may be written as an anonymous variable. Otherwise the presence of such *singleton* variables often indicates a misspelled variable name, and the compiler can give optional warning for them. The warning is suppressed for singleton variables the name of which begins with underscore (they can be used to improve readability of the source text).

Examples of variables:

```
X
X1
_13
VAR

_What_is_this
```

1.1.3 Compound terms

A compound term is of the form

```
term_name(arg1,arg2, ...,argN)
```

where **term_name** is an atom, and **arg_1**, **arg_2**, ... **arg_N** are arbitrary Prolog terms. **term_name** is called the **name** of the term, **N** is its **arity**. The **term_name/N** expression is called the **functor** of the compound term. **N** has to be not higher than 255.

This notation of a compound term is called functional notation. Every compound term has functional notation.

When its principal functor is an operator, a compound term has also another notation, the operator notation. This notation is used when reading and writing compound terms whose functor name is an operator. The notion of operators is described in section 1.1.5.

When the principal functor is **'.'/2** (the name of the functor is the atom containing one dot character, and the arity is two), then the compound term is called a list. Lists can be written in a third notation, called list notation (see the next section).

A term that does not contain any unbound variables is called ground terms.

Examples of compound terms:

```
foo(1,2)
bar(foo(apple),apple(foo))
.(1,[])
```

1.1.4 Lists

There is a special class of compound terms, whose functor is **'.'/2**. These terms are called lists and they can be written in a different, more readable format. The term

```
.(Head, Tail)
```

can be written as

```
[Head | Tail]
```

Head is called the **head** of the list and **Tail** is called the **tail** of the list. **Head** is the first element of the list. If the tail itself is a list too, then the list

```
[H1 | [H2 | T] ]
```

can be written as

```
[H1, H2 | T]
```

H2 is the second element of the list.

There is a special atom **[]** called the **nil** atom, which by convention indicates the empty list, the list without elements. If the **tail** of a list is the empty list, the vertical bar and the empty list can be omitted. So the following three expressions denote the same lists:

```
[a | [b | [] ] ]
[a, b | [] ]
[a, b]
```

If the tail of a list is a variable, the list is called **partial list**, if the tail of a list is not a list, it is called **not proper list**.

1.1.5 Operators

If a unary or binary functor (a functor with arity 1 or 2) is declared as an operator, then terms created using this functor can be written in a different format, in so called operator notation.

There are some predefined operators and the user can define others (see the built-in predicate **op/3** and directive **op**). An operator is defined by its name, specifier and priority. Any atom can be the name of an operator.

There are three classes of operators:

- prefix
- infix
- postfix

A unary functor can be declared as a prefix operator or a postfix operator. If **foo/1** is such a functor then the term

```
foo(ARG)
```

can be written as

```
foo ARG
```

or

```
ARG foo
```

depending on whether **foo** is a prefix or a postfix operator.

If **foo/2** is a binary functor which is declared an infix operator, the term

```
foo(ARG1, ARG2)
```

can be written in the form

```
ARG1 foo ARG2
```

The priority of an operator is an integer, in the range 1-1200. The lower is the priority the stronger binds the operator. The priority resolves the ambiguities arising if an expression contains more then one operator. E.g. the expression

```
ARG1 op1 ARG2 op2 ARG3
```

is parsed as

```
ARG1 op1 (ARG2 op2 ARG3)
```

that is equivalent with

```
op1(ARG1, op2(ARG2, ARG3))
```

if the priority of **op1** is greater than the priority of **op2**. But if the priority of **op1** is less than the priority of **op2**, then the same expression is parsed as

```
(ARG1 op1 ARG2) op2 ARG3
```

that is equivalent with

```
op2(op1(ARG1, ARG2), ARG3)
```

The specifier of an operator is a mnemonic that defines the class (**prefix**, **infix** or **postfix**) and the (**right-associative**, **left-associative** or **nonassociative**) of the operator. The associativity resolves the ambiguities arising if an expression contains more than one operator with the same priority. E.g., the term

```
ARG1 op ARG2 op ARG3
```

is parsed as

```
ARG1 op (ARG2 op ARG3)
```

when **op** is a right associative infix operator, and it is parsed as

```
(ARG1 op ARG2) op ARG3
```

when **op** is a left associative infix operator. If **op** is nonassociative operator, the expression above without parentheses is not a legal term; trying to read it in causes a syntax error exception.

The specifier names are:

Specifier	Class	Associativity
fx	prefix	non-associative
fy	prefix	right-associative
xfx	infix	non-associative
xfy	infix	right-associative
yfx	infix	left-associative
xf	postfix	non-associative
yf	postfix	left-associative

An argument with the same priority as a non-associative operator must be enclosed in parentheses. For example

```
fx fx Arg
```

expression (here **fx** is an operator with specifier **fx**) causes syntax error, it must be written as

```
fx (fx Arg)
```

There cannot be two operators with the same class and name, or an infix and postfix operator with the same name.

There cannot be a left bracket (see section 1.1.7 below) and a prefix operator with the same name. There also cannot be a right bracket and an infix or postfix operator with the same name.

The priority of the comma predefined operator cannot be changed. (Note that the comma token itself is not an atom, but it is treated as synonymous with the `','` operator in appropriate context.)

If the priority of an operator is not less then the priority of the comma predefined operator (1000), then an expression in operator notation with such an operator for its main functor appearing an argument (of another term written in functional notation, or in list notation) must be enclosed in parentheses. So the expression

```
assertz(head:-body)
```

causes a syntax error, because the priority of the **:-** infix operator is not higher (1200, numerically not less) than the priority of comma. The valid expression is:

```
assertz((head:-body))
```

An atom, which is currently defined as an operator name or bracket name, cannot be the immediate operand of an operator in a term written in operator notation (i.e., when it occurs just as an atom, not as operator). In such context the atom must be parenthesized. For example

```
Rel == '@<='
```

is invalid; the valid expression is:

```
Rel == ('@<=')
```

This restriction may cause some confusion, especially with alphanumeric tokens used as operator names (like '**is**').

1.1.6 The predefined operators

Priority	Specifier	Operators
1200	xfx	:-
1200	fx	:- ?-
1150	fx	dynamic
1100	xfy	;
1050	xfy	->
1000	xfy	','
900	xfx	as
900	fy	\+
700	xfx	= \= == \==
700	xfx	@< @=< @> @>=
700	xfx	=..
700	xfx	=:= =\= < =< > >=
700	xfx	is
600	xfx	: $:
500	yfx	+ - \/ /\
400	yfx	* / mod // rem
400	yfx	<< >>
200	xfx	**
200	xfy	^
200	fy	- \

1.1.7 Brackets

CS-Prolog II introduces the notion of user brackets. A bracket is defined by two atoms — bracket open and bracket close — and a priority. The atom formed by concatenation of bracket open and bracket close is the name of the bracket. An expression

br_open arg_1, arg_2, ...arg_N br_close

is equivalent with the following term

br_name((arg$_1$, arg$_2$, ... , arg$_N$))

(provided that **br_open**, **br_close**, and **br_name** are the respective atoms for a bracket). E.g., if **<:** and **:>** are declared as open and close atoms of a bracket then the expression

```
<: 1, 2, 3 :>
```

is equivalent with the term

```
<::>((1,2,3))
```

The priority of a bracket is an integer, in the range of 1-1200. The special priority value 1202 can be used too, with a special meaning. If a bracket is declared with priority 1202 then the bracket components function simply as a new set of parentheses. The other priority values are all equivalent. (Priority value 0 is used for removing a bracket declaration.)

There is one predefined user bracket declared in CS-Prolog:

Priority	Open	Close
1200	'{'	'}'

The priority of the predefined curly brackets cannot be changed.

In normal practice, however, the unquoted open-curly and close-curly tokens are used instead of the quoted atoms indicated above. The unquoted variant is defined by the Prolog Standard; the user brackets facility is a CSP-II extension. The unquoted and quoted variants yield the same result (the structure name in both cases is the special unquoted atom **{}**), but only matching pairs are accepted.

The components of a bracket definition (**br_open**, **br_close**, and **br_name**) must be distinct atoms, different from the empty string. No component can be involved in any other bracket definition.

There cannot be a left bracket (see section 1.1.5 above) and a prefix operator with the same name. There also cannot be a right bracket and an infix or postfix operator with the same name.

1.1.8 Tokens

The basic element in the Prolog source text, recognized by the compiler and by the term-input built-in predicates, is the character sequence known as a token. The variables, atoms, numbers are tokens. Some single characters such as open paren (`(`), close paren (`)`), open list (`[`), close list (`]`), open curly (`{`), close curly (`}`), head tail separator (vertical bar, `|`), comma (`,`) are also tokens in themselves.

Tokens are delimited by any character that cannot be absorbed into them (as white-space characters in most cases).

1.1.9 End token, end of file

The end token consists of the period character (`.`) followed by white space character or end of file character. The end token has a special meaning. Each term shall be terminated by it. The end token cannot be part of any term. The character representation of it in a source or data file is a period, but the input predicate **read_token/[1,2]** returns it as a special atom **end_of_term**.

There is another special atom **end_of_file**, which is returned by character or term reading input predicates if the file end is reached.

The atom consisting of a single dot character (`'.'`) must not be followed by white space text, since that denotes an end token.

1.1.10 Comments

A comment is a sequence of characters that is treated as a single white-space character by the compiler and it is otherwise ignored.

There are two kinds of comments: single line comments and bracketed comments.

A comment can contain any combination of characters from the character set except as noted below.

Single line comments start with the percent sign character (`%`) and finish with newline character; they cannot contain newline character inside.

Bracketed comments begin with a slash star character sequence (`/*`) and finish with the star slash character sequence (`*/`), they cannot contain this comment closing sequence. A bracketed comment can occupy more that one line.

1.1.11 Quoted strings

A sequence of items called quoted characters enclosed in a pair of matching quote characters constitutes a quoted string. There are three kinds of quoted strings, according to the three different quote characters available (single, double, and back quote character, see section 1.1.13). Single quoted strings constitute (single) quoted tokens that are atoms (as term). Back quoted strings in CSP-II are treated as single quoted strings (an extension allowed by the Standard). The handling of double quoted strings depends on the value of the Prolog flag **`double_quotes`** (see the next section and 1.2.1) at the time when the read-term or Prolog text is input.

The common properties of quoted strings are described in this section.

A quoted string token can be spread over two or more lines by means of continuation escape sequences. A quoted string containing one or more continuation escape sequences denotes the same token as the one obtained by removing the continuation escape sequences from the original quoted string.

The content of a quoted string is a sequence of so called quoted characters, which may be:

ordinary character (alphanumeric, graphic, solo, or non-alpha extended);
space character;
the quote character framing the quoted string, doubled (representing one such quote);
a non-doubled quote character, different from the quote framing the quoted string;
meta escape sequence;
control escape sequence;
octal escape sequence;
hexadecimal escape sequence.

The escape sequences mentioned above are described in section 1.1.14.

1.1.12 Double quoted lists

A quoted string framed by a pair of double quote characters (**"**) is a double quoted list token. It denotes a term which depends on the value of the Prolog flag **double_quotes** (see section 1.2.1) at the time when the read-term or Prolog text is input. Normally (by default) it is treated by the system as a character code list, i.e., a list which has the same number of elements as there are quoted characters in the back quoted string. The elements of this list are integers corresponding to the ASCII codes of the characters in the string. For example, the following two terms are equivalent:

```
"abcd efgh"
[97, 98, 99, 100, 32, 101, 102, 103, 104]
```

The handling of double quoted tokens can be modified by setting the **double_quotes** flag. Depending on the current value of this flag, a double quoted token is read as a list of characters or simply as an atom (single quoted token) instead of the default handling described above.

1.1.13 The CS-Prolog character set

The alphanumeric characters include the uppercase and lowercase letters of the English alphabet, the 10 decimal digits and the underscore (**_**) character. The alphanumeric characters are used to form atoms and variables. The set of alpha characters might be extended to include national letters, depending on the features and current settings of the operating system.

The white space characters are the space, tab, carriage return and newline characters. The space character denotes itself when quoted. An unquoted white space character is sometimes necessary to separate tokens, but is not itself a token or part of a token. Comments are treated as a single white space character; i.e., they are ignored except for the token-separation effect.

The graphic character set consists of the following characters:

#	Hash mark (number sign)
$	Dollar
&	Ampersand
*****	Asterisk
+	Plus sign
-	Minus sign
.	Period
/	Forward slash
:	Colon
<	Left angle bracket
=	Equal sign
>	Right angle bracket
?	Question mark
@	At sign (unit price)
~	Tilde
^	Caret

The graphic characters can be concatenated to form atoms. Two adjacent atoms must be separated by at least one white-space character when the last character of the first atom and the first character of the second atom

are graphic characters. The backslash (\) character, categorized as meta character, in graphic tokens also behaves like a graphic character.

The solo characters are the following:

(	Left parenthesis
)	Right parenthesis
[	Left bracket
]	Right bracket
{	Left brace
}	Right brace
|	Vertical bar (head tail separator character)
,	Comma
!	Exclamation mark or cut
;	Semicolon
%	Percent sign (end line comment character)

A solo character denotes itself when quoted. An unquoted solo character is a token in itself except that % and the remaining characters on the line are a comment. A solo character token need not be separated by white space from an adjacent token.

The meta characters are the following:

\	Backslash
'	Single quote
"	Double quote
`	Back quote

A meta character modifies the meaning of the character or characters following it. A backslash character starts an escape sequence in a quoted string and in a character code constant, but in a graphic token it behaves like a graphic character. The quote characters are used to indicate the start and the end of a quoted string of the corresponding kind.

In CSP-II other characters, known as non-alpha extended characters, also can appear as quoted characters and in comments. Any character supported by the platform and not included in one of the categories listed above is treated as non-alpha extended character if its code value is between 1 and 254. (Note that this is different from the Standard where each extended character must be assigned to one of the five categories above.)

1.1.14 Escape sequences

Quoted tokens, character code list tokens and character code constants can contain **escape sequences**. An escape sequence denotes a single character and always begins with a backslash (\).

Escape sequences are used to specify characters such as carriage return and tab movement, and to provide literal representations of nonprinting characters and characters having normally special meanings such as single quotation mark ('), double quotation mark ("), back quotation mark (`), and the backslash itself. (Note that the delimiting quotation mark character can be represented inside the quoted string also by doubling the character.)

The following escape sequences are valid in CS-Prolog:

Category	Escape sequence	Remark
control escape sequences:		
	\a	Alert (bell)
	\b	Backspace
	\f	Form feed
	\n	New line
	\r	Carriage return
	\t	Horizontal tab
	\v	Vertical tab
meta escape sequences:		
	****	Backslash
	\'	Single quotation mark
	\"	Double quotation mark
	\`	Back quotation mark
octal escape sequence:		
	\O..O	an ASCII character with the code in octal form
hexadecimal escape sequence:		
	\xH..H	an ASCII character with the code in hexadecimal form

In the last two lines ***O..O*** indicates octal digit sequence, ***H..H*** indicates hexadecimal digit sequence. Note that these two escape sequences are finished by a backslash.

It is an error if a backslash appears within a quoted string in a combination other then one of the cases above or in a continuation escape sequence (immediately preceding a newline character).

Using the octal or hexadecimal escape sequences any character from the ASCII character set can be specified. For example, the ASCII backspace character can be given by the normal **\b** escape sequence, or by **\10** (octal) or by **\x8** (hexadecimal) sequence.

As it was explained earlier, the backslash — newline character combination can be used as continuation escape sequence. When a newline character immediately follows the backslash, the backslash and newline will be ignored and the next line will be treated as part of the previous one.

1.2 Program structure

A CS-Prolog application program is composed of one or more modules. Each module has a name, and is compiled as one unit (i.e., parts of one module cannot be compiled separately and one compilation unit may not consist of more than one module). The final application program is constructed by the linker from individually compiled modules. There cannot be two modules with the same name in an application program.

One (and only one) of the constituent modules shall contain the goal of the program — an exported user-defined predicate that will be called by the runtime system. This goal has a fixed name **main_goal**. Running the application amounts to evaluating the main goal.

The functor of the main goal can be either **main_goal/0** or **main_goal/1**. If the predicate of arity 1 is defined, it will be invoked with a list argument composed by the runtime system from the command line argument words supplied by the user at invocation (see chapter 35). The predicate has to be made **public** (exported).

A module consists of a sequence of directives and predicate clauses (Prolog text). Syntactically both directives and clauses are read-terms, i.e., well formed terms followed by an end-of-term token. A module must begin with the module head directive, and may end with the optional module end directive or at the end of the input.

1.2.1 Directives

Directives provide declarative information to the compiler; they are not executable entities even if the syntax suggests so. They specify properties of the procedures defined or referenced in the Prolog text, or the format and syntax of read-terms in the Prolog text.

Directives (except the module head directive) may appear anywhere in the module. The scope of a directive begins at the directive itself and lasts to the end of the module, or, in some cases, to the nearest related occurrence of the same directive.

The functor of all directives is **`:-/1`**. The argument of the

```
:-
```

operator is a compound term. The arity of this compound term is usually 1; but there are some directives that take more arguments.

Several directives have the following generic syntax:

```
:-directive_name(PredicateIndicators).
```

A *predicate indicator* (often mentioned simply, but not quite precisely, as *functor*) is a compound term **`'/'(A,N)`** where **`A`** is an atom or a parenthesized atom, and **`N`** is an integer. It is usually written in operator notation as **`A/N`**, and indicates the procedure whose identifier is **`A`** and whose arity is **`N`**.

The **`PredicateIndicators`** argument can be a single predicate indicator, a non-empty list of predicate indicators, or a sequence of predicate indicators connected by the **`','/2`** operator (i.e., a comma-separated sequence enclosed in parentheses).

Here follows the complete list of directives of CS-Prolog. Some of them, related to modularity, are explained in more detail in chapter 1.2.3.

Module head directive

The module head directive must precede any other directive and clause in the source text.

```
:-module(Module_name, PredicateIndicators).
```

`Module_name` is an atom, different from **nil**. **`PredicateIndicators`** specify the exported predicates. An empty list also can be given but the compiler issues a warning in this case (There is no normal use for a module without exported procedures).

Module end directive

The module end directive is optional in CSP-II; it can be used to indicate the end of the source text for a module.

```
:-endmod.
```

If this predicate is not present then the module ends at the end of the input stream for the compilation. If present, it must not be followed by any directive or predicate clause.

Import directive

The import directive serves for declaring predicates to be imported. The primary function of the import directive is to determine the source module and the name and arity of the predicate to be imported. Its secondary function is to determine the local name under which the imported predicate is intended to be used.

```
:-import(Import_functor_desc_list).
```

`Import_functor_desc_list` is a single **`Import_functor_desc`**, a non-empty list, or a sequence of **`Import_functor_desc`**-s of the imported predicates.

The **`Import_functor_desc`** can have the following most general form:

```
[Mod_name:]ImportedPredicate [as Alias_predicate]
```

where **`Mod_name`** is either an atom or a variable, the **`ImportedPredicate`** is either a predicate indicator or a variable, and the **`Alias_predicate`** is either a predicate indicator or an atom. The brackets here mean that something is optional, that is, the module name part or the alias part or both of them can be omitted.

Dynamic directive

The dynamic directive serves for declaring predicates to be dynamic, so that the program can manipulate them (add, delete clauses) during runtime.

```
:-dynamic(PredicateIndicators).
```

The compiler assumes that functors of calls that do not match any definition in the module (a predicate definition or an import, foreign, or dynamic declaration) are functors of dynamic predicates. A warning, however, is issued by the compiler for these — unknown — calls, because in many cases unknown calls are result of clerical errors.

Operator directive

The operator directive declares the properties of the user-defined operators for the compiler (see section 1.1.5). These declarations have no effect during runtime.

```
:-op(Priority,Associativity,Operator_name_list)
```

About the meaning of the arguments, see the description of the **op/3** built-in predicate in part III.

Bracket directive

The bracket directive declares the properties of the user-defined brackets for the compiler (see section 1.1.7). These declarations have no effect during runtime.

```
:-bracket(Priority,Open_atom,Close_atom)
```

About the meaning of the arguments, see the description of the **bracket/3** built-in predicate in Part III.

Foreign directive

The foreign directive declares that the indicated predicates are written in a language other than Prolog.

```
:-foreign(PredicateIndicators).
```

The description of how to use the foreign language interface is described in chapter 37.

Meta predicate directive

The meta predicate directive gives the compiler information about the arguments used in the calls of a predicate (at the moment when the call is actually performed).

The form of the meta predicate directive is:

$:-meta\_predicate(name(M_1,M_2,...,M_n)).$

where M_1, M_2,..., M_n are mode-specifiers for the respective argument of the procedure. The value of a mode-specifier is one of the atoms: **+**, **-**, **?**, **:**.

The mode of an argument can be

+
: **input** — the argument should be instantiated (at least partially)

-
: **output** — the argument should be an uninstantiated variable

?
: **unknown** — the argument can be any term (either input or output)

:
: **procedure** — the argument is a callable term that may be called later (by a metacall) or some other object that needs module name expansion

The use of the meta predicate directive is not essential except when a predicate passed as an argument will be called in a meta call from another module, see section 2.6. However, it may improve the efficiency of the program (both in size and execution speed); and provides means for expressing and checking (now at runtime only) the intended usage of a procedure.

Note that importing a predicate does not mean importing its meta predicate declaration. The directive has to be repeated in the module where the predicate is imported. (Header include files can help in having the declarations in one place.)

Discontiguous directive

The clauses of a user-defined procedure should normally form a contiguous group in the Prolog text, i.e., clauses for different procedures should not be interspersed and directives should not appear inside the group. The enforcing of this rule helps the compiler detect some frequent errors, e.g., due to misspelling or omitted arguments. If, however, it is desirable for any reason to write a procedure in several non-contiguous parts, the discontiguous directive can be used to suppress this checking:

```
:-discontiguous(PredicateIndicators).
```

The directive specifies that each of the indicated predicates may be defined by clauses which are not consecutive read-terms of the Prolog text.

If Prolog text contains a discontiguous directive related to a predicate **`PI`**, then **`PI`** can be indicated in any number of other discontiguous directives in the Prolog text. The first such directive, however, shall precede all clauses for the procedure **`PI`**.

set_prolog_flag directive

The set_prolog_flag directive changes some options that influence the parsing of the source text during compilation. It implements a subset of the **set_prolog_flag/2** built-in predicate.

```
:-set_prolog_flag(Flag, Value).
```

Two flags can be changed for the compilation: **`double_quotes`** and **`float_range_checking_function`** (see chapters 16 and 25). The default setting of **`float_range_checking_function`** is **`underflow_exception_after_rounding`** for compilation, while for the runtime system it is **`denormalize`**.

1.2.2 User-defined predicates

User-defined predicates (procedures) consist of a sequence of clauses that have the same principal functor. For a call, which has this functor, the first clause is tried, and on backtracking the others are retried. The order of clauses is defined by the order in which the clauses appear in the Prolog text. (This is true also for the statically compiled portion of dynamic predicates.)

All the clauses for a user-defined procedure shall be consecutive read-terms in the Prolog text (not interspersed with directives or clauses for other procedures) unless there is a discontiguous directive for the procedure in the Prolog text.

If no clauses are defined for a predicate indicated by a dynamic directive or a discontiguous directive, then the procedure is regarded as existing (dynamic), but has no clauses initially. The compiler issues a warning if the predicate is indicated only in a discontiguous directive.

If there are calls for a user-defined procedure in the text, but the procedure has no definition whatever, then the compiler issues a warning and the procedure is regarded as dynamic, but non-existing.

The difference between an existing dynamic procedure with no clauses and a non-existing one is that in the first case a call to the procedure simply fails, while the call to a non-existing procedure is under control of the **unknown** Prolog flag.

A clause is a non unit clause (rule) or a unit clause (fact). Every clause in a module has to be terminated by a period (**end token**).

A unit clause is a callable term (an atom or a compound term which is not a list). It defines a fact in Prolog parlance. A unit clause

```
Fact.
```

expresses that **`Fact`** is true (holds). The principal functor of a fact is its functor itself.

Non unit clauses are of the form

```
Head :- Body.
```

where **Head** is a callable term and **Body** is a call sequence. Such a clause is customarily referred to as a rule. The informal meaning of this rule is: **Head** is true if **Body** is true. The principal functor of a rule is the principal functor of its head.

A call sequence is a call or a sequence of calls separated by commas. A call sequence is true if each call in it is true (see also conjunction in the next section).

A call can be:

- a call sequence (containing multiple calls)
- a group of alternatives (or simply group)
- an if-then control construct
- an if-then-else control construct
- one of the atomic control constructs **!/0** (cut), **true/0**, **fail/0**
- a control construct **call/1**
- a prefixed call
- a callable term
- a variable (metacall)

A group is a sequence of alternatives separated by semicolons, or a single sequence. A group is true if one of its alternatives is true. (In Prolog execution the alternatives are tried one by one during subsequent backtracks.) Each alternative is a call sequence except that an if-then construct can appear only as the last alternative. This is because **(;)/2** (semicolon) serves two distinct functions depending on whether or not the first argument is a compound term with functor **(->)/2**. When it is, then we have an if-then-else construct instead of group. (See also disjunction in the next section.)

A prefixed call is a compound term with functor **(:)/2**, usually written in operator notation as **ModName:Inner**, where **ModName** should be a valid module name by the time of the activation, and **Inner** is a call. The prefixed call construct instructs the system to take the definition of **Inner** from module **ModName** (see also chapter 2).

A callable term is an atom or a compound term which is not a list. Let **Func** denote the functor of the callable term. If there exists a unit clause in the program with functor **Func** that is unifiable with the call, then the call is true (succeeds). If there exists a rule (non unit clause) in the program with functor **Func** and the head of the rule is unifiable with the call, and the body is true then the call is true (the call succeeds). The built-in predicates are conceptually present in any module; they succeed or fail depending on their arguments and the actual program state.

The variable constituting a metacall has to be instantiated by the moment of the invocation. Its value can be any valid call except a variable. The effect of a metacall is the same as calling the **call/1** control construct with that variable as the argument.

The control constructs are described in the next section.

Note that if a call in a call sequence is itself a (multi-member) call sequence (enclosed in parentheses in operator notation) then the effect is the same as for the flattened structure obtained by removing these parentheses.

Example:

```
:- module(relatives,[main_goal/0]).

mother(kate, mary).                          /* fact */
father(kate, joe).                           /* fact */
mother(christine, mary).                     /* fact */
father(christine, joe).                      /* fact */
father(sylvester, emeric).                   /* fact */
mother(sylvester, elizabeth).                /* fact */
parent(Child,Parent):-                       /* rule (head)*/
      mother(Child, Parent);                 /* alternative */
      father(Child, Parent).                 /* call */
sibling(X,Y):-                               /* head */
      parent(X,PARENT), parent(Y,PARENT).    /* call sequence */
main_goal:-                                  /* rule (head) */
      sibling(kate, christine).              /* call */
```

This small program will successfully terminate. If the main goal is changed to

```
main_goal:-
       sibling(kate, sylvester).
```

then the program will terminate with failure. With the main goal changed to

```
main_goal:-
       sibling(kate, WHO), write(WHO).
```

the program will produce the output **christine**.

If the module head is changed to

```
:- module(relatives,[main_goal/1]).
```

and the **main_goal/0** clause is replaced by

```
main_goal([WHO]):-
       sibling(kate, WHO).
```

then the program expects exactly one command line argument, and succeeds or fails depending on whether or not the argument supplied is the name of a sibling of **kate**.

1.2.3 Control constructs

Note in advance that **(;)/2** serves two distinct functions depending on whether or not the first argument is a compound term with functor **(->)/2** (disjunction or if-then-else).

See also Control predicates, chapter 26.

call/1

```
call(Goal)
```

is true if and only if **Goal** is instantiated to a valid call other than a variable by the time of the invocation, and **Goal** is true.

When **Goal** contains **!** (cut) as subgoal, the effect of **!** does not extend outside **Goal** (i.e., **call/1** is non-transparent to cut).

!/0 — cut

Cut is used to control backtracking. The call

```
!
```

always succeeds (is true). On backtracking into it, this call modifies the execution so that no choices are permitted between the call and its parent. Parent of the cut is usually the invocation of the procedure containing the cut as call, but some control constructs are non-transparent to cut, or have non-transparent parts (see if-then, if-then-else, and **call/1**). Meta calls, being equivalent with **call/1**, are also non-transparent to cut. In such cases the parent is the nearest non-transparent call (call sequence).

(‘,’)/2 — conjunction

```
','(First, Second)
```

or written in operator notation as

```
First ',' Second
```

is true if and only if **First** is true and **Second** is true.

Note that in most cases the unquoted variant of the comma infix operator also can be used.

Call sequences mentioned in the previous section are constructed using conjunction.

(;)/2 — disjunction

```
;(Either, Or)
```

or written in operator notation as

```
Either ; Or
```

is true if and only if **Either** is true or **Or** is true.

Note that in a disjunction **Either** cannot be a compound term with **(->)/2** as principal functor. Such a complex construct is interpreted rather as if-then-else (see below). An if-then construct can be used in **Either** position only indirectly, embedded in a **call/1** construct, or transformed to an equivalent but syntactically different call sequence. For example

```
a -> b ; c
```

is an if-then-else construct. A disjunction in which the **Either** part is equivalent with **(a -> b)** can be written as

```
(a -> b), true ; c
```

The difference between the two is that in the first case if **a** succeeds, then **c** is not tried on backtracking, while in the second case **c** will be tried. In both cases **a** can succeed at most once.

Groups of alternatives mentioned in the previous section are constructed using disjunction.

(->)/2 — if-then

An if-then construct is of the form:

```
If_part -> Then_part
```

(operator notation). Here **If_part** and **Then_part** both are call sequences. The meaning of this construct is the following. The construct is true if **If_part** is true and then **Then_part** is true for the first solution of **If_part**. On backtrack the eventual alternatives (choice points) in **If_part** are ignored.

If_part is non-transparent to cut, i.e., a cut appearing in it cuts only choice points opened during the evaluation of **If_part** itself, as if **If_part** were executed separately in a **call/1** construct.

(;)/2 — if-then-else

An if-then-else construct is of the form:

```
;(->(If_part, Then_part), Else_part)
```

or written in operator notation:

```
If_part -> Then_part ; Else_part
```

The main functor of the construct is **(;)/2** and the main functor of the first argument is **(->)/2** (c.f. disjunction above). Here **If_part**, **Then_part**, and **Else_part** all are call sequences. The meaning of this construct is the following. The construct is true if either **If_part** is true and then **Then_part** is true for the first solution of **If_part**, or **If_par**t is false and **Else_part** is true. On backtrack the eventual alternatives (choice points) in **If_part** are ignored.

If-then-else constructs are often used in a chain, and can be regarded informally as a committed choice in the sense that the first **If_part** that succeeds in such a chain reduces the continuation to the associated **Then_part**.

If_part is non-transparent to cut, i.e., a cut appearing in it cuts only choice points opened during the evaluation of the **If_part** itself, as if **If_part** were executed separately in a **call/1** construct.

Note that (;)/2 and (->)/2 are predefined operators so that

```
(If -> Then ; Else)
```

is parsed as

```
;(->(If,Then),Else)
```

true/0

fail/0

`true/0` always succeeds; `fail/0` always fails. In trace mode, however, they behave as equivalent built-in predicates (i.e., they appear in the trace output).

2. Modules

Modules are individually compiled parts of a CS-Prolog program. The main purpose of using modules is to divide large programs into units of manageable size, with clearly defined interfaces. The module concept also facilitates information hiding (encapsulation) and reusability.

In a module there can be calls invoking predicates defined in that module, built-in predicates, and predicates imported from other modules. A predicate is defined in a module if there is a static definition, or a dynamic declaration, or foreign declaration for it. A predicate can be imported only if it is specified as public in the supplier module. Non-exported static predicates are hidden inside the module, so identically named local procedures in different modules do not interfere with each other.

2.1 Visibility rules

The data names (atoms) are global for the entire program; there are no local atoms in a module or in a process.

A source/sink stream (see chapter 3) is global in the sense that if a stream is opened then any process in any module can read from / write to it using its identifier or alias.

Value names (see **set_value/2** built-in predicate in chapter 19) are also global for modules, but not for processes. That means that a value name is associated with the same value if it is used in different modules by the same process. On the other hand, different processes can and will have different values associated to the same value name, or not have a defined value for the name at all.

The operator and bracket declarations are local for a module in compile time. It is important to know that any operator or bracket declaration given statically in a module in form of **op/3** or **bracket/3** directive, has effect only during the compilation of that module. The runtime system is initiated only with the built-in operators and brackets. So if you want an operator or bracket to be declared during the execution, it has to be added by the **op/3** or **bracket/3** built-in predicate at the beginning of the program execution. The operator and bracket definitions are global data in the system.

The dynamic database of a module is visible for any other module via the clause retrieval and modification built-in predicates, prefixing the predicate name with the supplier module name in the client module (see module prefixing in section 2.3). Similarly, asserting to, and deleting from, an other module's dynamic database is also possible. However, a module has to import the dynamic predicates if they are directly called, like in the case of static predicates.

There are some predefined standard modules in the system that define some of the built-in predicates.

2.2 Exporting from a Module

Exporting procedures (predicates) from a module means that any other module will be able to use them via explicit or qualified import. The exported predicates are listed in the module head directive.

The module head directive specifies the name of the module and the public predicates that are exported from it. Its format:

```
:- module(ModName,PredicateIndicators).
```

`ModName` must be an atom (the name of the module) different from **nil**. **`PredicateIndicators`** is a list or sequence of predicate indicators being exported. The predicates represented in this list must be known within the module at compile time, i.e., they must be either defined statically, or declared as dynamic or foreign predicates. Imported predicates also can be re-exported. The exported predicates are called **public**. E.g.:

```
:- module(list_handling,[member/2,append/3]).
```

When a single predicate is exported, the second argument can be given as a single predicate index. The argument also can be given an empty list, which causes a warning message only.

2.3 Module prefix

If a predicate is referenced in a module and the definition of the predicate is in another module then the referenced predicate name must be prefixed by a module prefix. The module prefix supplies the name of the module which defines or re-exports the predicate. Prefixing is usually written in operator notation; the colon built-in infix operator **(:)/2** serves for this purpose:

```
Mod_name : Term
```

This module prefix is used for instance if a dynamic clause is accessed, that is defined in another module:

```
assertz(mod1 : fact)
abolish(mod1 : fact/0)
```

Another example of using module prefixes is the qualified import that is described in the next sections.

2.4 Predicate import

In order to make it possible to call a predicate from another module, the predicate has to be imported. The functors of the imported predicates are normally enumerated in import directive(s). An import directive has the form:

```
:-import([Modname:PI, ... ]).
```

The exact syntax of import directive is described in section 1.2.1. **Modname** is the name of a module where the procedure characterized by predicate indicator **PI** is defined. This predicate has to be declared public in module **Modname** (by placing its indicator into the export list).

The module prefix (the module name and the colon operator) can be omitted, or **Modname** can be specified as a (placeholder) variable, to indicate non-restricted import. In this case the system will search all modules for the predicate during link editing or binding. The linker issues a warning if a matching predicate is exported from more than one module, and resolves the reference to the first matching predicate encountered.

To avoid name clashes or to be able to assign alias name to an imported predicate there is the possibility of giving a local name to an imported functor. The

```
:-import([ModName:AlienName/Arity as LocalName...]).
```

directive allows us to use **LocalName** inside the module instead of the original **AlienName** in calls of the **AlienName/Arity** predicate. Usually **AlienName/Arity** is a normal predicate indicator. The next section describes the case when **AlienName** and/or **ModName** are variables.

LocalName can be a predicate index or an atom. If it is an atom, then it is conceptually augmented with **Arity** to form the local predicate index **LocalName/Arity**. Otherwise, the arities must be the same.

A module prefixed call

```
Mod : Call
```

can be invoked even if there is no import directive for the functor of **Call**. It behaves as if an import directive were declared for the functor of the call while compiling this call, but the import declaration does not remain permanent; other occurrences of calls with the same functor have to be module prefixed again. Note also, that, unlike the import directive, a prefixed call will not conflict with existent local definitions of the predicate indicator. In other words, prefixed calls make it possible to call a predicate from another module directly even if the same predicate indicator has a local definition, too.

2.5 Flexible predicates

It is possible to import an unspecified predicate and refer to it in the module by a local name (late binding). The form of such an import is

```
:-import([X as LocalPred]).
```

or

```
:-import([ModName:X as LocalPred]).
```

X is a variable (an identifier beginning with an upper case letter or underscore). This variable is not a real Prolog variable, it will never have any value; it is rather only a placeholder to show that the imported predicate functor is not fixed at compile time.

`LocalPred` is the (full) predicate indicator used locally to call the imported predicate. It defines a so-called flexible predicate, which can be used in clauses, but the program must provide an actual predicate functor for it before the first call is executed. This can be done by **bind/2** built-in predicate. If ModName is given as a name of a module, then only predicates defined in that module can be assigned by binding. If ModName is given as a (placeholder) variable or the module prefix is missing then any compatible predicate is accepted.

Binding of a flexible predicate is process-specific, i.e., any modification in the state of a flexible predicate performed in one process is not visible for other processes.

Note that the predicate, which is supplied as the actual binding, does not have to be exported; it also may be any compatible local procedure in the module where **bind/2** is called. The important condition is that both arguments of **bind/2** must be visible at the place where the call is issued.

The binding so created can be subsequently changed by another **bind/2** call, or can be destroyed by an **unbind/1** call. A call of a currently unbound flexible predicate raises an exception, regardless of the current value of the **unknown** Prolog flag.

2.6 Predicates with callable arguments

Some predicates have input arguments the final purpose of which is to be called as metacall during execution of the predicate. Let's call these predicates **generic** predicates. E.g. the **'\+'/1** or **catch/3** built-in predicates are called with argument(s) that are callable terms, and are invoked as a metacall later. When a metacall is performed, it is interpreted always within the module of the metacall. So there is a problem if a **generic** predicate is exported and is called from another module since the metacall will not find the called procedure in its own module. The call of a generic predicate has to append information to callable arguments about the module where these arguments are defined as procedures. By this it will be ensured that, when the argument appears subsequently as a metacall, the appropriate procedure definition can be located. This is done by the so-called module name expansion mechanism.

When calling a generic predicate, the argument(s) are expanded, the module name is prefixed to them. But for this purpose instead of the

:

infix operator the

$:

infix operator is used. The meaning of this operator is similar to the meaning of the normal module prefix operator, but makes it possible to call non-public predicates, too.

For those built-in predicates that need it, module name expansion is done automatically by the system. For user-defined predicates, the expansion can be achieved by an explicit declaration of a **meta_predicate** directive. If an argument is declared as a **procedure argument**, the system will expand the argument whenever the predicate is called. E.g. the **'+\'/1**, and **catch/3** built-in predicates have automatically the following meta_predicate directive:

```
:-meta_predicate(\+(:)).
:-meta_predicate(catch(:,?,:)).
```

If a predicate is imported and it has a **meta_predicate** declaration in the module where it is exported, this declaration has to be repeated in the importing module, too.

3. Input/Output System

3.1 Sources and sinks

A source/sink is a basic notion when speaking about input/output. A program can output results to a sink or input data from a source.

An example for a source/sink can be a file or the user's terminal. Each source/sink is associated with a sequence of bytes that can be read from it or that were written out to it. A specific source/sink (possibly a file name) can be given as argument to the **open/[3,4]** predicate, and later all subsequent references (read or write) are made using an internal representation or an alias (see section 3.3).

3.2 I/O modes

An I/O mode is an atom defining the I/O operations that may be performed on a source/sink. This atom is an argument of **open/[3,4]**. CS-Prolog supports the following I/O modes:

`read`
: The **read** mode is used if the source/sink is a source. If it is a file, it already has to exist.

`write`
: The **write** mode is used if the source/sink is a sink which is initially empty. If it is an already existing, the previous contents will be lost.

`append`
: The **append** mode is used only if the source/sink is a sink. When it is an existing file, output starts at the end of that file. When the sink is not an existing file, this I/O mode is equivalent with **write**.

3.3 Streams and aliases

A stream identifier identifies a stream during a call of an input/output predicate. It is a ground term created when a source/sink is opened by a call of **open/[3,4]**. The actual form of this term is not important because it cannot be generated by the user.

Any stream may have an associated stream alias. A stream alias is an atom, which may be used to refer to that stream. An alias can be created at the time when the stream is opened, and the association is automatically destroyed when the stream is closed. A particular alias refers to at most one stream at any one time. All built-in predicates that have a stream identifier as an input argument accept a stream alias for that argument as well. However those predicates which return a stream identifier in an argument never return or accept a stream alias (such a predicate is e.g. **current_input/1** or **stream_property/2** etc.).

3.4 Standard streams

Three streams, the standard input, standard output and standard error stream, are predefined and cannot be closed. The standard input stream has the alias **user_input**, the standard output stream has the alias **user_output**, and the standard error stream has the alias **user_error**. These streams are set up by the operating system for the CS-Prolog runtime. By default, all the three are connected to the control terminal, unless redirected.

3.5 Current streams

During execution, there is always a current input stream and a current output stream. When an input predicate does not have an explicit stream identifier argument, it reads from the current input stream. Similarly, when an output predicate does not have an explicit stream identifier argument, it writes to the current output stream.

By default, the current input and output streams are the same streams as the standard input and output streams, but the built-in predicates **set_input/1** and **set_output/1** can be used to change them.

When the current input stream is closed, the standard input stream automatically becomes the current input stream. Since the standard input stream cannot be closed, this guarantees that the current input stream always refers to an open stream. The case with output is similar.

3.6 Memory streams

A memory stream is a stream created by **open/4** using a special open option (see section 3.8); it is a source/sink that resides in the memory. During execution the program can write to it, then reopen it with **reopen/3**, and then read from it. It can be useful for converting terms to atoms and vice versa, or for simply storing temporary information.

The contents of a memory stream is lost after the stream is closed or after the termination of the program.

3.7 Stream positions

A stream position is a ground term, which identifies an absolute position within the source/sink to which the stream is connected. It is maintained by the system and can be obtained using the predicate **stream_property/2**.

At any time, the stream can be repositioned by calling **set_stream_position/2**.

Not all streams can be repositioned; disk files and memory streams can.

On a platform where binary and text file formats differ, repositioning a text file can give unexpected results. (Repositioning to the beginning of file is always safe.)

3.8 Options on stream creation

When a stream is created with **open/4** certain options can be specified in an option list argument. The permitted options are:

`type(T)`
: **`T`** = **text** or **binary**. Specifies whether the stream is a text stream or a binary stream. Default is **text**.

`reposition(Bool)`
: **`Bool`** = **true** or **false**. When **`Bool = true`** checks if it is possible to reposition the stream to a previously visited position. If this is not possible for the particular source/sink, **open/4** raises an exception.

`alias(A)`
: **`A`** is an atom. Specifies that the atom **`A`** is to be the alias for the stream. **open/4** raises an exception if this alias already refers to an open stream.

`eof_action(Act)`
: **`Act`** = **error**, **eof_code** or **reset**. The atom **`Act`** specifies the effect of attempting to read past the end of a stream. If it is **error**, an **existence_error** exception is raised, signifying that no more input exists in this stream. If **`Act`** = **eof_code**, the normal end-of-stream marker is returned, i.e. **end_of_file** for term input and character input, and **-1** for byte input. If **`Act`** = **reset**, the stream is reset and another attempt is made to read from it. This is likely to be useful when reading from a device such as a terminal. This action sets the stream not to be in **past_end_of_stream** state and may also perform some operation to reset the source to which the stream is attached. Default for **eof_action** is **error**. This option is valid only for input streams.

`stream_type(F)`
: **`F`** = **file** or **memory**. Specifies the type of stream. Default is **file**.

`line_len(N)`
: **`N`** is an integer. Specifies the maximum line length for a sink. If a write predicate outputs a term and the output length would exceed **`N`**, the term is split into two or more lines. This option is valid only for output streams and has effect only on term output (**write** built-in predicate family).

If the option list contains conflicting options, the rightmost one applies.

The following three are legal open options but contain redundant information, so they are rarely used (possibly only for verification). If any of these options is incompatible with the value of another argument of **open/[3,4**], an exception is raised.

`input`
: The stream is a source. It can be specified if the I/O mode is **read**.

`output`
: The stream is a sink. It can be specified if the I/O mode is **write** or **append**.

`file_name(N)`
: **N** is an atom. **N** is the file name of the source/sink. Its value has to be the same atom as the first argument of **open/[3,4**]

`mode(M)`
: **M** is an atom, the I/O mode of the source/sink. Its value has to be the same atom as the second argument of **open/[3,4**]

3.9 Stream properties

Various properties of streams can be accessed via the predicate **stream_property/2**. The

```
stream_property(Stream, Property)
```

call returns various stream-property pairs. The values of **Property** can be the following:

`file_name(F)`
: When the stream is connected to a source/sink which is a file or a memory stream, **F** is the name of the file which is the source/sink for the stream. (In case of memory streams, it is a virtual name.) The standard streams have file names **user_input**, **user_output**, **user_error**.

`mode(M)`
: **M** is unified with the I/O mode specified when the source/sink was opened; its value can be **read**, **write** or **append.**

`alias(A)`
: If the stream has an alias, then **A** is that alias.

`position(P)`
: If the stream has the **reposition(true)** property, **P** shall be matched with the current stream position of the stream.

`reposition(B)`
: If repositioning is possible on the stream then **B** is unified with **true**, else **B** is unified with **false**.

`type(T)`
: The value of **T** defines whether the stream is a text stream (**T** = **text**) or a binary stream (**T** = **binary**).

`input`
: This stream is connected to a source.

`output`
: This stream is connected to a sink.

`stream_type(T)`
: The value of **T** defines whether the stream is a standard stream (**T** = **standard**), a file stream (**T** = **file**), or a memory stream (**T** = **memory**)

`eof_action(Act)`
: For sources, it returns value of **eof_action** option specified when opened the source.

`line_len(N)`
: For sinks, it returns **N** = 0 if the stream was not open providing a **line_len** open option, otherwise returns the value of the open option.

`line_count(L)`
Unifies **L** with the number of lines read in or written out.

`line_position(P)`
: Unifies **P** with the number of characters read in or written out in the last line.

`char_count(N)`
Unifies **N** with the total number of characters read in or written out.

3.10 Read options

A read options list is a list of various options, which affect the read_term/ [2,3], and **tread_term/3** predicates. The options are:

`variables(Vars)`
After reading a term, **Vars** is matched with the list of the variables in the term, in left-to-right traversed order. Anonymous variables are included in the list.

`variable_names(VN_list)`
After reading a term, **VN_list** is unified with a list of elements where each element is a term

```
V = A
```

and **V** is a named (non-anonymous) variable of the term, and **A** is an atom whose characters are the character of **V**. The anonymous variables (the variables denoted by '_') are not included.

`singletons(VN_list)`
After reading a term, **VN_list** is unified with a list of elements where each element is a term

```
V = A
```

V is a named variable of the term which occurs only once in the term, and **A** is a (quoted) atom whose characters are the characters of V (i.e. the quoted version of the token **V**). The anonymous variables (denoted by '_') are not included.

3.11 Write options

A write options list is a list of various options, which affect the **write_term/[2,3]** predicates. The options are:

`quoted(Bool)`
Bool = **true** or **false**. When **Bool** is **true**, each atom and functor is quoted if this would be necessary for the term to be read as data by **read_term/3**.

`ignore_ops(Bool)`
Bool = **true** or **false**. When **Bool** is **true**, each compound term is output in functional notation. Neither operator notation nor list notation is used when this option is in force.

`numbervars(Bool)`
Bool = **true** or **false**. When **Bool** is **true**, a term of the form **'$VAR'(N)**, where **N** is an integer, is output as a variable name consisting of an upper-case letter possibly followed by an integer. The upper-case letter is the **i+1**-th letter of the alphabet, and the integer is **j**, where

```
i = N mod 26
j = N / 26
```

The integer **j** is omitted if it is zero. For example, **'$VAR'(1)** is written as **B**, or **'$VAR'(26)** is written as **A1**.

`maxdepth(N)`
Compound subterms at nesting level **N** will be output as **'*'**. The tail of a list longer than **N** elements will be output as **'...'**.

`variable_names(VN_list)`
VN_list has to be a list where each element of this list is a term of form

V = A

V is a variable contained in the term to be output, and **A** is an atom. When writing out the term each occurrence of **V** will be replaced by the characters of **A**.

The first three options can also be given in short form, without argument. The

```
quoted, ignore_ops, numbervars
```

option specifications are equivalent, respectively, with

```
quoted(true), ignore_ops(true), numbervars(true)
```

3.12 Reaching end of stream

At the end of a stream, it is still valid to call an input predicate that returns a specific value to indicate that end of stream has been reached. The character code and the byte input predicates (**get_code/[1,2]** and **get_byte/[1,2]**) return the integer **-1**. The character and term reading predicates return the atom **end_of_file**.

When one of these terminating values has been read, the stream is said to be in state **past_end_of_stream**. It depends on the value of the open option **eof_action** what happens if input is performed on a stream, which is **past_end_of_stream** (see section 3.8).

3.13 Text and binary streams

There are two types of streams: text streams and binary streams. There is a separate set of predicates that input or output data for each of these types. To read or write data to a text stream the character and the term input/output predicates can be used, for binary streams the byte input/output predicates can be used.

The type of a stream can be set by a stream option when opening the stream (see section 3.8). Every stream has a property showing its type.

A text stream consists of a sequence of characters. Characters are represented in CS-Prolog as one-char atoms. For every character there is a number between 0 and 255, the ASCII code of the character which is called character_code.

The set of characters extended with the **`end_of_file`** atom is called **in_character**; the set of character_codes extended with the number **`-1`** is called **in_character_code**

A binary stream consists of a sequence of bytes. A byte is a number between 0 and 255. The set of bytes extended with the number **`-1`** is called **in_byte**

3.14 Character and term input

There are two different ways of getting input from a text source. A source can be considered as a

- sequence of characters (atoms consisting of one character),
- sequence of Prolog tokens and Prolog terms

The different input built-in predicates - the **get_char** or **get_code** family and the **read** family - can be used in mixed way for the same stream. That means that after reading a byte from a source the program can perform character input and then read term form the same stream.

The term input can be done in CS-Prolog in two ways. The first of them - with predicate **read_term/3** and its relatives - is the traditional **read** predicate of Prolog languages. It consists of reading tokens from the selected source until a period (**end token**) is found. Then forming a valid Prolog term from the tokens is attempted, and the **end token** is discarded. Therefore, in this case every term to be read in has to be terminated by an **end token**. The following character sequence does not constitute a legal input for a **read_term/3** (supposing that **`hello`** is not a postfix operator):

```
foo + coo hello.
```

The other set of read predicates consists of the **tread_term/3** and its relatives. These predicates try to read tokens from the input as long as they form a valid Prolog term. If a token is found in the stream that cannot be included into the term, then the reading stops, leaving that token in the stream. For example, if the input stream contains the characters shown above, **tread_term/3** would read the term

```
foo + coo
```

leaving the token **`hello`** in the stream. But, as this token had to be physically read from the stream, if a byte or character input is performed later, it will read characters from the stream that are after the token **`hello`**. This token will be included in the consequent term input.

So mixing the **tread_term**-like and the character (or byte) input can produce unexpected results. E.g. as it is explained in the previous paragraph, a **get_char** can return a character, when several characters before it in the source (that form the token read ahead) remain unread. Similarly the **line_count**, **line_position**, **char_count** and **position** stream properties return the real state of the source stream, not taking into consideration that some of them did not yet appear in Prolog level.

4. Exception handling

During the execution of a program various exceptional situations can occur. Such events happen if an error is detected in a built-in predicate, or external interrupt is signaled from the operating system, or a system resource gets exhausted (memory, disk space), and so on. If such a situation takes place and there is no method provided to handle it, the execution of the program stops, possibly writing out an error message.

The exception handling mechanism of CS-Prolog gives the possibility for the programmer to deal with exceptional situations so that in most cases the program can recover from errors without outside intervention.

An artificial error situation can be provoked by calling a specific built-in predicate. With an appropriate exception handler, it can be used to jump out from the middle of an execution flow (see **throw/1**).

4.1 Format of error terms

The exception handler is informed about the error event. Every error or interrupt is represented by a Prolog term. A user-defined error term can be an arbitrary Prolog term, but the errors and interrupts signaled by the runtime system have standard format. The standard error terms are the following:

```
instantiation_error
type_error(ValidType, Culprit)
domain_error(ValidDomain, Culprit)
existence_error(ObjectType, Culprit)
permission_error(Operation, PermissionType, Culprit)
representation_error(Flag)
evaluation_error(Error)
consistency_error(ConflictType, Culprit)
syntax_error(SyntErrAtom)
resource_error(Resource)
interrupt(InterruptName, InterruptData)
system_error
clp_system_error
```

The meaning of the arguments of an error term is the following.

`Culprit`

The term which caused the error. This is usually an argument of the erroneous goal, but can also be a subterm of an argument. For consistency error and certain types of domain error **`Culprit`** is a pair (of two items connected by '**+**') involved in the conflict. In the special case of trying to read past the end of a stream, the **`Culprit`** is the atom **past_end_of_stream**.

`Flag`

One of the following atoms, indicating the type of a representation error:

```
character
character_code
in_character_code
max_arity
max_integer
min_integer
max_atom
max_format
max_message
max_term_size
max_path_in_clause
max_call_in_clause
max_call_in_path
max_clause_complexity
```

`ValidType`

One of the following terms. They indicate the legal term type for the culprit.

```
variable
constrained_variable
atom
integer
float
compound
list
```

```
atomic
byte
callable
character
evaluable
in_byte
in_character
number
predicate_indicator
table_key
periodicity
clp_expression
clp_evaluable_expression
clp_solver_selector
```

ValidDomain

One of the following terms. They indicate the range where the culprit should have been in.

```
character_code_list
close_option
flag_value
io_mode
non_empty_list
not_less_than_zero
positive_number
operator_priority
operator_specifier
prolog_flag
bracket_priority
read_option
source_sink
stream
stream_option
stream_or_alias
stream_position
stream_property
write_option
unique_name
channel_mode
radix
maxdepth
timeout
queue_size
net_init_mode_option
net_close_mode_option
net_partner_limit
option
net_config_option
explicit_netobj_name
ip_addr
ip_port
netconfig_action
net_attr_value
(un)conditional
mediator_kind
hnms_option
bracket_name_component
module_name
proper_list
ground_term
clp_relation_functor
clp_linear_expression
clp_evaluable_linear_expression
valid_formula_for_clp_solver
installed_clp_solvers
```

ObjectType

One of the following atoms indicating the type of the culprit causing the error:

```
procedure
source_sink
stream
module
event
```

```
process
term
channel
control
netobject
nethost
netservice
```

Operation

One of the following atoms indicating the type of the action that caused the error:

```
access
create
input
modify
open
output
reposition
remove
parallel
bind
unbind
```

Error

One of the following atoms indicating the type of the arithmetic error encountered:

```
float_overflow
int_overflow
undefined
underflow
zero_divisor
bad_float
bad_format_item
```

PermissionType

One of the following atoms indicating the type of the culprit for which an invalid operation was requested:

```
binary_stream
flag
operator
past_end_of_stream
control_construct
built_in_procedure
private_procedure
static_procedure
source_sink
stream
text_stream
process
channel
backtrackable_procedure
nonbacktrackable_procedure
local_proc
non_flex
part_flex
wrong_procedure
would_create_loop
event
value
netobj_name
netobject
net_attribute
net_buffering_limit
bracket
```

Resource

One of the following atoms indicating the type of the resource that has been exhausted:

```
memory
disk_space
open_streams
```

SyntErrAtom

An atom denoting indicating the type of the syntax error.

ConflictType
One of the following atoms indicating the type of the conflict detected:

```
netaddr
selfhost
identical_bracket_name_components
```

InterruptName
One of the following atoms indicating the type of the interrupt:

```
user
program
task_imeout
```

InterruptData
Any term

For each category above where a taxative enumeration is given, an atom beginning with the prefix **unknown_** can also appear (e.g., **unknown_interrupt_name**), which indicates an internal error in the runtime system.

4.2 Additional information term on exceptions

There are several important data items not included in the error terms, which may be needed for an appropriate exception handler. These items are:

Goal
The goal which was executed when the exceptional situation occurred. The special value, the integer 0, is given as **Goal** in the case where the system is not able to supply the Goal or where there is no one particular Goal that caused the error. It contains a module prefix indicating the module where the error occurred.

ArgNo
An integer in the range **0..max_arity** indicating which argument of **Goal** caused the error. **ArgNo** = 0 means either that no one particular argument caused the error, or that the system is not able to determine which argument caused the error.

Other
A list containing other relevant information. In most exceptions cases its value is the empty list.

These terms are combined into a list forming the so-called **error info** . It is provided by the system for the exception handler together with the error term. The **error info** is a list that has at least two arguments:

```
[Goal, ArgNo | Other]
```

4.3 Error terms

In the description of error terms, we will refer to the elements of the error info term: **Goal**, **ArgNo** and **Other**.

4.3.1 Instantiation error

```
instantiation_error
```

Goal is not sufficiently instantiated for the call. It means that the **ArgNo**-th argument or a subterm of the **ArgNo**-th argument is an unbound variable, and it is not legal. If **ArgNo** is ambiguous its value can be zero.

Examples:

```
X is 1 + Y.
instantiation_error, (error_info = [mod $: _ is 1+_, 2])

functor(_, _, 20).
instantiation_error, (error_info = [mod $: functor(_,_,20), 0])
```

4.3.2 Type error

```
type_error(ValidType, Culprit)
```

The type of the **ArgNo**-th argument or the type of a subterm of the **ArgNo**-th argument is incorrect. A type error is never caused by a term being an unbound variable, that is an instantiation error. A type error can be caused by a term being a non-variable where a variable is required.

Examples:

```
assertz(42).
type_error(callable, 42)

X is a + 12.
type_error(number, a)

open(file, read, name).
type_error(variable, name)
```

4.3.3 Domain error

```
domain_error(ValidDomain, Culprit)
```

The type of the **ArgNo**-th argument is correct but the value is outside the range for which the predicate is defined. Also may occur when a named option is processed (either of the form **keyword(value)** in an option list, or **keyword** and **value** as separate arguments of the call) and **value** is not allowed for **keyword**. In this case **Culprit** has the form **keyword+value**.

Examples:

```
arg(-1, a(1,2), A).
domain_error(not_less_than_zero, -1)

op(1300, xfy, #).
domain_error(operator_priority, 1300)

port_current_attribute(_, advertised, yes).    % Allowed values: on, off !
domain_error(net_attr_value, advertised+yes)
```

4.3.4 Existence error

```
existence_error(ObjectType, Culprit)
```

The object on which an operation is to be performed does not exist.

Examples:

```
undef(arg). /* undef/1 predicate is not defined) */
existence_error(procedure, undef/1)

open(missing_file, read, S).
existence_error(source_sink, missing_file)
```

4.3.5 Permission error

```
permission_error(Operation, PermissionType, Culprit)
```

It is not permitted to perform the specified **Operation** on the specified **Culprit**, which has type **PermissionType**.

Examples:

```
assertz(var(1)).
permission_error(modify, static_procedure, var/1)

get_char(user_output, C).
permission_error(input, stream, user_output)
```

4.3.6 Representation error

```
representation_error(Flag)
```

An error in trying to represent the result of a computation. Some limit has been exceeded.

Examples:

```
char_code(C,566)
representation_error(character_code)
functor(T, a, 1000).
representation_error(max_arity)
```

4.3.7 Evaluation error

```
evaluation_error(Error)
```

Operands of an evaluable functor are such that the operation has an exceptional value.

Examples:

```
X is 65536 * 65536.
evaluation_error(int_overflow)
X is log(-1).
evaluation_error(undefined)
```

4.3.8 Consistency error

```
consistency_error(ConflictType, Culprit)
```

Two distinct operands or sub-operands in an operation are incompatible with each other, although each of them is valid in itself. This error can occur in manipulating complex system objects, mainly in connection with networking.

Examples:

```
community_change_config(add_private, [[self, hostname(foo)]]).
consistency_error(selfhost, self+foo)
```

The error indicates that **`foo`** is not known as being the symbolic name (or one of the aliases) of the host computer where the program is running.

4.3.9 Syntax error

```
syntax_error(SyntErrAtom)
```

Characters that should form a Prolog term are syntactically invalid. It can occur while executing the term input procedures (**read_term/3**, **tread_term/3**).

4.3.10 Resource error

```
resource_error(Resource)
```

The runtime system ran out of some resource while executing. This resource can be for instance the memory allowed for the CS-Prolog system or the disk space when writing out to a file.

4.3.11 System error

```
system_error
```

A system error arises only when the runtime system encounters some error situation, which does not correspond to any of the other error classes. In most cases, this arises either in interactions with the operating system, or because of some error in the organization of the program. If possible, the **Other** element of the error info term gives some available information about the system error.

4.3.12 Interrupt

`interrupt(InterruptName, InterruptData)`

Some event from the outer world interrupted the execution. This interrupt can come from the operating system (user interrupt), or from an other part of CS-Prolog (time-out, interrupt caused by another process).

The user interrupt is routed to the **main process** in the first place. If it is already in terminated state, then the system is stopped with a corresponding error message.

Examples:

Interrupting the execution with Ctrl-C will cause

`interrupt(user, [])`

exception in execution of the main process.

4.3.13 CLP System error

`clp_system_error`

This error can occur only in a system with CLP extension enabled, and only when a solver is active. It indicates some internal error detected by the CLP organizer module or an active solver. The **Other** element of the error info term gives some available information about the error, either in textual or in numerically coded form.

4.4 Error handling procedures

There are two ways to handle an error in CS-Prolog. The first (predicate **catch/3**) is simpler to use, but it offers less possibilities to define the action to be taken in case of error. The other (with predicate **protected/3**) is more complicated, but allows using techniques that are more sophisticated. The two possibilities can be used together in the same program.

Both predicates establish a safeguarded environment for a goal, and can deal with exceptions raised during the execution of this goal. During the execution of a protected goal, new protected environments can be set up with subsequent calls of **catch/3** or **protected/3**. These nested environments will be referred to as **protection levels**. When an exception arises at run time, the system searches the ancestors of the current call to find a call of **catch/3** or **protected/3** predicate. The nearest suitable one will determine what type of exception handling is invoked. These methods are described in the next sections. If the handling fails, the exception is passed to the next outer protection level, that is, to the nearest ancestor **catch/3** or **protected/3** call. The outermost level is defined by the run-time system, outside the user program. The system level error handling consists of writing out a standardized message and stopping with error. If the handling succeeds, it also defines some corrective action to be taken.

Note that the subsequent descriptions are conceptual only; the actual implementation is somewhat more complicated because the two distinct protection mechanisms are merged into one nesting chain.

The user can also raise any exception via the **throw/1** predicate. As a special case, a standard exception term can be thrown which will be indistinguishable from the corresponding exception raised by the system.

4.5 Error handling with catch/3

To create a protected environment for a call, invoke it through the **catch/3** predicate:

```
catch(Goal, Catcher, Recovery)
```

The system sets up the protection environment, and begins evaluating **`Goal`**. If Goal succeeds or fails, so will the **catch/3** call, too. If, however, any error is signaled during the execution of the **`Goal`**, and the error is not handled at an inner level, then the control returns to the **catch/3** predicate. The possible variable bindings and backtrackable side effects that occurred while executing **`Goal`** are undone, as if **`Goal`** had **fail**-ed. **`Catcher`** is then unified with the (copy of the) error term, and if the unification succeeds then the **`Recovery`** call is executed (as a replacement for the original **catch/3** call). Otherwise, if the unification fails, then the exception cannot be handled at this level, and so it is passed to the next protection level (to the nearest **protected/3** or **catch/3** ancestor, or to the system level handler).

The execution of **Recovery** is not protected from exceptions by this catch. If we want it to be protected, it has to be a call of **catch/3** (or **protected/3**) itself.

Examples:

```
catch(Goal, Y, true).
```

This call protects the **Goal** call from every error. If during the evaluation of **Goal** an exception is raised, the execution will continue with successful termination of the **catch/3** call.

```
catch(Goal, existence_error(file,FN), inform(FN)).
```

This call will handle all existence errors raised during the execution of **Goal** if the error was caused by a non-existent file. The execution will continue with calling the **inform/1** user defined procedure with the name of the missing file.

4.6 Error handling with protected/3

The idea of error handling by this method is the following: a goal may be executed in a protected environment. When an exception occurs during the execution of this goal, the nearest exception handler will be executed. If this handler fails, that means that it is not prepared to cope with the error, so a higher level protection must deal with the situation instead. In order to handle the exception the handler must succeed. It passes then two items to the system: a replacement goal, and a decision about which call is to be replaced by it. There are three choices:

Replace the erroneous or interrupted call ('on the spot' - not always possible);

Replace the call of the original goal (preserving its safe environment) at this level of protection;

Replace the entire protected call removing this level of protection.

To establish this kind of protected environment for a goal **Goal**, call it using the **protected/3** built-in predicate:

```
protected(Goal, Handler, UserTerm)
```

It executes **Goal** and if during the execution an exception happens, then the system calls the **Handler/5** predicate supplying it with five arguments as explained below. (Let's suppose the value of **Handler** is **handler_name**)

```
handler_name(ErrorTerm, ErrorInfo, UserTerm, UnwindChoice, Result)
```

If the call of the exception handler fails, the exception will be propagated to the next higher level of protection. If the exception handler succeeds, it gives the information how to continue the execution. The meaning of arguments supplied to the error handler is as follows:

ErrorTerm is the error term.

ErrorInfo is the additional error information list (see section 4.2)

UserTerm is the third argument of the appropriate **protected** call (in its current instantiation state); it can be used to pass additional data to generalized handlers.

UnwindChoice and **Result** are output arguments. The system invokes the **handler_name** procedure with unbound variables in the output (fourth and fifth) argument positions. If the call succeeds, then **Result** is expected to be bound to a callable term that will be called after the handler terminates — it replaces the interrupted call, or the goal argument of its appropriate **protected** ancestor, or the entire **protected** ancestor, depending on the value of **UnwindChoice**. **UnwindChoice** = 0 (number zero) means that the handler handles the error 'on the spot', where the exception occurred. **UnwindChoice** = 1 (number one) means replacing the call of the goal argument of this protected with **Result** and executing it. **UnwindChoice** = 2 means replacing the entire protected call with **Result**, deleting this level of protection. Any other value of **UnwindChoice** causes a system error.

If **UnwindChoice** is not zero, the possible variable bindings and backtrackable side effects that occurred while executing **Goal** (first argument of **protected**) are undone ('unwound'), as if the **Goal** had **fail**-ed.

Note that handlers usually are evaluated in the environment where the error occurred, but the replacement goal can be executed after unwinding several levels. For this reason, instead of the **Result** term supplied by the handler, a systematically renamed copy of that term is used as the replacement if unwind is requested.

Practically this means that if any unbound variable from **Goal** and/or **UserTerm** occurs in **Result** too, then it is replaced by a new unbound variable (detached from the original).

As it was explained, protected environments may be nested. In order to avoid an infinite cycle of repeated exception handling, any exception occurring during the execution of a handler will be passed to the next higher protection level, including the case when any of the output arguments are incorrect.

If an interrupt arrives to a process that is executing an exception handler, the interrupt is delayed until the actual exception handling is finished. This means, that the interrupt will cause an exception only when the original (protected or unprotected) environment is restored. If an exception handling process takes too long time in a program where many interrupts are caused for the actual process, it can produce an interrupt queue overrun error.

There are some severe errors, which cannot be handled on the spot. For these, before the handler is invoked the system unwinds the innermost protected environment. These severe errors are:

```
resource_error(_, memory, [stack])
resource_error(_, memory, [heap])
resource_error(_, memory, [trail])
system_error(_, [mode_in])
system_error(_, [mode_out])
```

The first three are raised when one of the main memory areas is full. The **mode** errors are signaled when an argument of a procedure is declared as input or output argument (see the **meta_predicate** directive, section 1.2.1) and the current value contradicts to this declaration.

4.7 Signaling errors

A program may artificially signal error conditions. Using the **throw/1** built-in predicate, it is possible to make the system act as if a given error had occurred. The error term will be the argument of the **throw/1** predicate; it can be an arbitrary Prolog term.

This possibility can be used to jump out from the execution of a goal if something has gone wrong. For example, let's suppose that

```
catch(do_something, my_break, true)
```

is called, and we are executing the **do_something** goal, a

```
throw(my_break)
```

call is invoked. In this case the system will abandon the execution of **do_something**, and will continue with successful termination of this **catch/3** call.

4.8 Error handling example

The following program demonstrates the use of error handling methods of CS-Prolog.

In this example, the execution is protected with a **catch/3** from any error. If an exception occurs the error message is written out and the program is re-executed. The steps of execution are protected by a **protected/3** predicate. The handler will catch the user interrupt, and the user can determine what to do, continue, skip one step and continue with the next or break the execution. For other interrupts, the handler calls a specific goal to deal with the event and than continues with the execution of the interrupted call. Finally, the handler catches the file operation errors where a non-existing file name is given. The user is prompted to enter the corrected file name and the program retries the file operation. Other exceptions are not handled, so they are propagated to the higher level **catch/3**.

```
program :-
        catch(execute, ERR, recover(ERR)).

recover(ERROR):-
        inform_on_error(ERROR),
                program.

execute:-
        is_end, !;
        protected(one_step,s_hand,0),
                execute.
```

```
s_hand(interrupt(user,_), [Call|_], 0, Unw, Res):- !,
       write('Interrupted, choose '), nl,
              write('      c      -continue'), nl,
              write('      s      -skip this step'), nl,
              write('      else   -break'), nl,
              get_char(Answ),
              (Answ = 'c', /, Unw = 0, Res = Call;
               Answ = 's', /, Unw = 1, Res = true;
               fail).
s_hand(interrupt(IName,IData), [Call|_], 0, 0, Call):- !,
       handle_interrupt(IName,IData).
s_hand(existence_error(file,FN),[C,1|_] 0, 0, Call):-
       replace_filename(C,FN,Call), !.

replace_filename(C,FN,Call):-
       write('Non existing file '), write(FN), nl,
              write(' enter new file name: '),
              get_line(New_FN),
              C =.. [FU, _ | Rest],
              Call =.. [FU, New_FN | Rest].

one_step:- ....

handle_interrupt( ...

is_end :- ...

inform_on_error( ...

...
```

5. Preprocessor

The CS-Prolog compiler incorporates a preprocessor. It is a clone of the usual preprocessors for the C language, with the difference that Prolog operators are used in the expressions for conditional compilation.

Only the compiler accepts preprocessor commands; there is no input procedure that would recognize such constructs at runtime.

The advantage of use of a preprocessor is that using macros the source can be made more readable, more accurate. Using header (include) files makes the interface between different modules cleaner, conditional compilation (**ifdef**) makes possible to keep in one source several versions of the same program, and there are many more benefits. It has to be mentioned, however, that this technique (natural in the C language) is not customary in Prolog. It can be confusing for instance to have a macro name with capital letters denoting an atom, because in the source it can easily interpreted as a variable. We hope though that the advantages of a preprocessor are worth to incorporate it in the system.

The preprocessor accepts directives that begin always with a **#** (hash mark) character, preceded only by whitespace on the line. The following sections describe the specific directives.

5.1 Macros

```
#define name token-string
```

Replace subsequent instances of **name** with **token-string**. After the replacement, the preprocessor re-starts to scan the **token-string** to expand eventual macros in it.

```
#define name(arg, ..., arg ) token-string
```

(Notice that there cannot be spaces between **name** and the open parenthesis.) Replace subsequent instances of **name** followed by an open parenthesis, a list of comma-separated sets of tokens, and a close parenthesis with **token-string**. Each occurrence of an **arg** is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded token-string unchanged. After the entire token-string has been expanded, the preprocessor re-starts its scan for names to expand at the beginning of the newly created token-string.

```
#undef name
```

Cause the definition of **name** (if any) to be forgotten from now on. No additional tokens are permitted on the directive line after name.

Examples:

```
#define line_length 80
```

Using this macro, if in the source there is somewhere the **line_length** atom it will be replaced with the number **80**.

```
#define INCR(x,x1) x1 is x+10
```

With this macro definition the operation of increasing a value can be standardized in the program. If later it is needed to increase by 20 instead of 10, only the macro definition has to be changed.

5.2 Include files

```
#include "filename"
#include <filename>
```

Include at this point the contents of **filename** (which will then be scanned by the preprocessor). When the **<filename>** notation is used, **filename** is only searched for in the standard places. Standard directories can be given setting the environment variable **CSPINC** to a colon separated list of directory paths. (See the -I option of the compiler, chapter 33.) When the **"filename"** notation is used, **filename** is searched first in the current working directory and, if it is not found, then in the standard places. No additional tokens are permitted on the directive line after the final " or >.

Include files can be useful when different modules contain the same code fragment (e.g. some common macro definitions). This directive can be nested, that is include files can contain include directives.

5.3 Conditional compilation

The basic construct of conditional compilation is a conditional block built according to the following scheme:

#if *condition...* (or #ifdef or #ifndef - see later)

if-section-body

#elif *..condition* (any number of occurrences, including 0)

elif-section-body

#else (optional - 0 or 1 occurrences)

else-section-body

#endif

The block consists of at least one section and a closing **#endif** directive. Each section begins with a section-control directive, which is followed by any number of source lines constituting the section body, up to the next section or the block end. The effect of the construct is that during preprocessing the whole block is either removed or is replaced by one of the section bodies, depending on the result of the sequential evaluation of the section control directives. After replacement preprocessing continues from the beginning of the substituted section body; after removal - immediately behind the removed block.

Conditional blocks can be nested up to 16 levels deep.

The result of the evaluation of a section control directive is an integer value. The first section for which the evaluation yields a non-zero result is selected for replacement.

The value of the **#if** and the **#elif** directives is the result of the evaluation of the *condition* part of the directive after macro symbol substitution. The value of the **#else** directive is always 1.

The *condition* is a special constant expression. After macro symbol substitution it must be a syntactically correct expression built from integers and atoms as primary (terminal) expressions. Compound expressions can be built like expressions for arithmetic evaluation using operators, parentheses, and a special function **defined** (c.f. Arithmetic evaluation in PART III), with the following differences:

- The set of allowed operators is different;
- Relation operators can also be used in subexpressions, not only at the outermost level;
- Relation operators and logical operators yield integer value 1 when they hold and 0 otherwise;
- Atoms remaining in the expression are regarded as undefined macro symbols and during evaluation they are replaced by the integer value 0 except in the argument position of the function **defined**.
- The evaluation is performed in 32-bit integer representation;
- Some operators are represented in more then one form (to allow for intuitive meaning, and to resemble C preprocessor directives).

The following basic operators are defined (in order of decreasing priorities):

\+ -	(logical *not*, prefix arithmetic negation)
* / << >>	(multiplication, integer division, arithmetic left shift, arithmetic right shift)
- + \/ /\	(subtraction, addition, bitwise *or*, bitwise *and*)
< <= == >= > \=	(arithmetic ordering relations)
&& ||	(logical *and*, logical *or*)

Alternative operator symbols accepted:

//	(for /)
=:= =	(for ==)
=\= \==	(for \=)
=<	(for <=)

Logical (truth) values are coded as 1 for *true* and 0 for *false*. Arithmetic operators are evaluated as usual. Arithmetic relation operators yield coded truth-values according to whether or not the respective relation holds.

Logical operators implicitly transform their operands before taking effect into coded truth-values. (0 for zero, 1 for non-zero value).

The special function **defined**(*name*) results in 1 (*true*), if *name* has been the subject of a previous **#define** without having been the subject of an intervening **#undef**, otherwise it yields 0 (*false*).

The effect of the individual conditional compilation directives according to the general scheme is as follows:

```
#if constant-expression
```

Lines following this directive up to the nearest associated directive **(#elif**, **#else**, or **#endif** at the same level) will be preprocessed (and compiled) if and only if **constant-expression** evaluates to non-zero.

```
#ifdef name
```

This directive is equivalent with **#if defined (name)**

```
#ifndef name
```

This directive is equivalent with **#if \+defined (name)**

```
#elif constant-expression
```

An arbitrary number of **#elif** directives is allowed between a **#if**, **#ifdef** or **#ifndef** directive and the nearest associated **#else** or **#endif** directive (explained below). The lines following the **#elif** directive up to the nearest associated directive (**#elif**, **#else**, or **#endif** at the same level) will be preprocessed (and compiled) if and only if the test directive (**#if**, **#ifdef** or **#ifndef**) starting the conditional block and all intervening **#elif** directives evaluate to zero, and **constant-expression** evaluates to non-zero. If **constant-expression** evaluates to non-zero, all succeeding **#elif** and **#else** sections within this conditional block will be ignored.

```
#else
```

The lines following this directive up to the associated **#endif** (at the same level) will be preprocessed (and compiled) if and only if the test directive starting the conditional block and all associated intervening **#elif** directives evaluate to zero. No additional tokens are permitted on the directive line.

```
#endif
```

Marks the end of both the conditional block and its last section. No additional tokens are permitted on the directive line.

Example:

```
#if defined(dos) \/ defined(os2)
#define MAX_FILE_NAME_LEN 8
#if defined(dos)
#define BAT_EXTENSION '.bat'
#else
#define BAT_EXTENSION '.cmd'
#endif
#elif defined(unix)
#define BAT_EXTENSION ''
#define MAX_FILE_NAME_LEN 44
#endif
```

This code fragment will set some parameters - numbers and atoms - that can be used later in the program, depending on the operating system the module is compiled for.

5.4 Predefined macro symbols

The following macro symbols are defined by the calling environment when the preprocessing starts:

`__FILE__`
: Contains the path name of the current input file (from which the line containing the macro reference had been read).

`__LINE__`
: Contains the line number (within the current input file) of the input line where the reference is located.

`sun`
: This macro is defined with the value 1 if the preprocessor is running on a Solaris platform. On other platforms the macro is initially undefined.

`linux`
: This macro is defined with the value 1 if the preprocessor is running on a Linux platform. On other platforms the macro is initially undefined.

`freebsd`
: This macro is defined with the value 1 if the preprocessor is running on a FreeBSD platform. On other platforms the macro is initially undefined.

5.5 Preprocessor command line options

The preprocessor recognizes some command line options. A macro can be defined on command line, and the standard library path list can be changed (where the include files are searched).

These options are described in details in chapter 33.

PART II

Parallel programming & real-time features

6. Introduction

The CS-Prolog system provides possibility to write, develop and run parallel Prolog programs. Independently from the fact that the host machine and/or operating system has the ability to execute more than one program at a time, the CS-Prolog system makes possible to develop and execute parallel programs based on its own capabilities.

The parallelism of the CS-Prolog is defined in the language itself and does not rely on any information about the underlying machine architecture. However, the CS-Prolog system tries to gain the advantage of the architecture of the machine on which the CS-Prolog system is installed. Since these pieces of information are hidden from the user, CS-Prolog programs are portable. It is the CS-Prolog system's task to distribute the parallel components in the most efficient way on any individual architecture and organize the communication among the processors.

The CS-Prolog system also provides possibilities to develop and execute applications that respond promptly to events generated explicitly either outside or inside the CS-Prolog system or implicitly by an inner clock. This feature enables the user to write real-time applications.

All the parallel and real time features of CS-Prolog are implemented via built-in predicates.

7. Basic notions

7.1 Processes

The basic notion of the CS-Prolog's parallelism is the process. Process is defined as the execution flow of a Prolog goal. The progress of a process is assumed independent from the execution of other processes. Every process has its own Prolog execution environment and dynamic database. Note that, conceptually, global values and flexible imported predicates also belong to the dynamic database. Each process can access only its own dynamic database, which is strictly separated from the other's databases. If a dynamic procedure has statically compiled clauses, then an individual copy of these clauses is created for each process when the process is started. In contrast, the static part of the program (the set of statically compiled procedures the process uses during the execution of its goal, the clauses of which cannot be modified) may be partially or totally identical with the static part of other processes. For example, processes may execute the same goal. Normally the static databases are identical; in the case of real multiprocessor architecture, however, there is a possibility to load different programs on different processors. These programs form a single application by sharing such system-wide common entities as channels and events (see later).

The separation of dynamic databases ensures that CS-Prolog processes may have influence on each other only by the CS-Prolog provided communication techniques (message, interrupt, event passing), or through external objects like files, or through system-wide common resources like Prolog flags.

CS-Prolog programs always execute processes. Even the simplest CS-Prolog program executing a single goal (the traditional case) appears as the execution of a main process.

Processes are identified by system-wide unique names. The number of processes is limited only by the amount of the available memory.

CS-Prolog provides the following two kinds of processes:

— self-driven (normal) process
— event-driven (real time) process

The self-driven process is the more usual kind of process, that's why we will call it also a normal process. A self-driven process is characterized by its goal. The execution of the self-driven process's goal is initiated simply by its presence in the CS-Prolog system, i.e. once a self-driven process has been created it will begin the execution of its goal as soon as it gets the control. The non-fatal termination of a self-driven process is also determined by the (either successful or unsuccessful) termination of its goal. At the moment of its termination the self-driven process disappears from the CS-Prolog system and will never reappear.

The event-driven process is also called real time process. A real time process is characterized by one goal for the initialization, one goal for the event handling, and by the description of the events that trigger the execution. The execution of the initializing goal begins as soon as the process gets the control. During the execution of the initializing goal the real time process acts as if it were a normal process. At the (successful) end of the initializing goal the real time process changes to a cyclic behavior. From this moment, the real time process is controlled by the incoming events and it executes its event-handling goal as answers to them. Whenever an event occurs the real time process performs its event-handling goal and consumes that event. This action is repeated for all incoming events (the repetition is organized by backtracking). A real time process is intended to answer continuously to events and it is not necessary to assume a theoretical termination of it. A real time process terminates if the execution of its event-handling goal fails. Such termination is considered as regular; it does not affect the overall success or failure of the application.

7.2 Phases in process creation

The process creation method in CS-Prolog is somewhere between the dynamic and static creation. To explain that, the lifetime of the CS-Prolog program is divided virtually into two consecutive phases:

— prelude phase
— working phase

When the CS-Prolog program (and the prelude phase) is started, the main process is started automatically. The goal of the main process is always **main_goal/[0,1]**. Only the main process and only in the prelude phase has

the right to create processes using the **new/[2,3]** and **new_rt/[5,6]** built-in predicates or to delete previously created processes using the **kill/1** built-in predicate. In the prelude phase the created processes are not alive yet. This means that all the necessary information is stored, but the execution of the other processes has not yet begun. The created processes become alive when the CS-Prolog system turns from the prelude phase into the working phase. Until then the creation and deletion of processes is dynamic.

The end of the prelude phase and the beginning of the working phase is indicated by the invocation of the **start_processes/0** built-in predicate.

When the CS-Prolog system gets into the working phase the previously created processes become alive and begin working as soon as they get the control. The main process remains alive and will behave like other self-driven processes. In the working phase there is no more possibility to create or delete processes and that's why the state of the processes in this phase is static. The only change in the set of processes is the automatic deletion of the terminated processes.

In those cases when the user does not want apply the parallel feature of CS-Prolog (there are no user defined processes) the working phase may be empty. However, in order to assist memory handling, it is advisable to invoke a **start_processes/0** call at the beginning of the program's work, notifying the scheduler that it does not have to reserve resources for process creation. This makes the prelude phase almost empty and the single process will run entirely in the working phase. This technique is not obligatory, but resource handling will be more efficient this way.

7.3 Processors

Since the number of physical processors and the connection among them are machine dependent resources, they are hidden from the CS-Prolog user. That's why the CS-Prolog language does not refer to the notion of processors. Instead, it defines the notion of virtual processors. The number of virtual processors is independent of the number of physical processors present in the system. A CS-Prolog program may refer to unlimited number of virtual processors. Virtual processors are identified by system-wide unique names and they play role only in the process distribution algorithm. The process distribution algorithm is an internal algorithm of the CS-Prolog scheduler, which distributes the created processes among the physical processors. When a new process is created (using the **new/3** or **new_rt/6** built-in predicate) the user can determine the virtual processor which the new process is intended to be delegated to. The only condition that the CS-Prolog scheduler ensures is that processes having the same virtual processor destination will be surely delegated to the same physical processor. No other condition is ensured, e.g. it is not sure that processes having different virtual processor destinations will be delegated to different physical processors. In other words, virtual processors define process groups the member processes of which shall run on the same physical processor.

If the user does not specify a virtual processor identifier for a process when it is created then the CS-Prolog scheduler decides on its own where to delegate the process.

7.4 Termination of CS-Prolog programs

A non-parallel CS-Prolog program (having no user-defined processes) terminates successfully or unsuccessfully if the execution of the main goal (the goal of the main process) was successful or unsuccessful, respectively.

A parallel CS-Prolog program (having at least one user-defined process) terminates if all of its processes have terminated. The termination of the whole CS-Prolog program is successful if all of its normal (self-driven) processes have terminated successfully. If at least one such process failed, the whole program is regarded as terminated unsuccessfully.

If the CS-Prolog program contains at least one real time process then this fact may reflect the aim of the user to write a program which is intended to work (theoretically) forever and in this case there is no reason to speak about the termination of the program. However, in order to avoid the necessity of using the cancellation methods of the operating system in these cases, the CS-Prolog runtime system will terminate a real time process when the event handling goal (or the initializing goal) of the process fails. Consequently, the event handling goals of real time processes should always succeed, except when the user wants the process to terminate. Note that when a real time process is terminated because its event handling goal failed, this termination is regarded to be successful.

If a run-time error is raised during the execution of any process or some other exception is directed to a process, and the error or exception is not caught by any protection level of that process, then the execution of the whole application is terminated and the control returns to the operating system.

A special exception dealing with termination should also be mentioned here: if the scheduler finds all processes waiting for some event, and decides that none of these events can possibly happen, then a *deadlock* exception is signaled to the main process.

The whole application can also be terminated by the **halt/[0,1]** predicate issued by any of its processes.

7.5 Channels and messages

The CS-Prolog system defines the notion of channels as a tool for the message passing between processes. There is no other way to pass messages. A channel is a directed pipe between processes (i.e. a single channel can only serve for one way message passing). It may appear and disappear dynamically during the program's lifetime. Channels do not belong to any process; instead, they act as system-wide available resources. Each channel has two ends: one for sending and one for receiving. Processes can own only the ends of channels and not the channels themselves. However, in the following description we will often say that the process "owns the channel", meaning only that the process holds the appropriate channel end. A process cannot hold both ends of a channel at the same time.

Channels are identified by system-wide unique names. The total number of channels in the system and the number of the channels a process can own are unlimited.

A message can be any Prolog term except a single unbound variable. Compound terms containing unbound variables are allowed.

7.6 Message passing

The passing of messages through channels is synchronized, i.e., the real message transfer takes place only when processes at both end of the channel are ready to send or receive the message, respectively. (It is called rendezvous or handshake method.) The message is effectively passed by copying the message term from the sender process to the receiver process and no unification on the unbound variables (if any) is executed at all. All messages have to be accepted. Since the **receive/[2,3,4]** predicate can have only an unbound variable as its message argument, the process can not reject a message directly, only after removing it from the channel.

The synchronous message communication system is coupled with interrupt handling. This means that whenever a process is waiting in a long–term communication predicate (which depends on the behavior of some other process), it might be interrupted and the operation will behave as a transaction (either completed totally or no effect at all). The exception from this rule is the broadcast style **send** operation (with more than one destination channels). When such a send is interrupted, it is possible that some of the recipients will have already received the message (and proceed with execution), while others remain waiting as if the sender hadn't even started the interrupted send operation. On this basis, it is better to consider such broadcasting send not as an elementary operation but instead as a (hidden) cycle of elementary operations (where the order of individual transfers is indeterministic).

7.7 Events

The basic notion of the real time programming is the event. Events serve for triggering real time processes (event driven processes). When a real time process is created then a non-empty set of events is assigned to it. This real time process will be sensitive to these events. This means that if one of these events occurs then the real time process launches its event-handling goal and consumes the event. For every real time process, the incoming events are gathered in a separate first-in-first-out input queue.

The events in CS-Prolog may arise from the following sources:

events generated explicitly by built-in predicates,

events generated explicitly by the external environment of the CS-Prolog system,

events generated implicitly by the internal clock of the CS-Prolog scheduler.

The number of the available events and the number of events that real time processes can be triggered for are unlimited. Events are identified by system-wide unique names.

Every occurrence of an event may have an optional data argument. The data argument may provide some additional information about the event. The event data is an arbitrary Prolog term, except the case of a single unbound variable.

Events generated implicitly by the internal clock of the CS-Prolog scheduler have priority over regular events. In the extreme, fast occurring timer events can delay indefinitely other event processing for the process concerned.

8. The scheduling mechanism

The CS-Prolog scheduler is part of the CS-Prolog run time system; its main tasks are:

to distribute the created processes among the available physical processors;

to control the quasi-parallel execution of the processes working on the same physical processor;

to handle the system-wide unique names (virtual processors, processes, channels, events);

to maintain other system-wide common resources, namely the set of Prolog flags and the tables of the current operator and bracket definitions;

to supervise the communication between processes (channel handling, message and event passing);

to monitor the global deadlock situations;

to handle the termination of the program and to return to the operating system.

8.1 The process distribution

During the prelude phase of the program, the CS-Prolog scheduler collects the user-defined processes created by the **new/[2,3]** or **new_rt/[5,6]** built-in predicates. Until the end of the prelude phase, processes can be removed by the **kill/1** built-in predicate. At the end of the prelude phase, the set of processes becomes fixed. The execution of the **start_processes/0** built-in predicate performs the process distribution algorithm. The CS-Prolog scheduler decides to which physical processor to delegate the individual processes and performs the delegation. After the distribution, the **start_processes/0** built-in predicate terminates successfully and the scheduler begins the execution of the created processes on each physical processor.

8.2 The parallel execution

Processes on different processors work independently and they have no influence on each other unless some communication takes place.

Considering one of the physical processors, the scheduler works on it as follows:

If there was no process delegated to it, it remains idle during the whole execution.

If there was only one process delegated to it, the single process owns the entire processor, so it will not be disturbed by other processes.

If there was more than one process delegated to it, then the scheduler runs them in a quasi parallel way (time sharing method). This means that the scheduler shares the processor among the involved processes assuming equal priority for all of them.

Since the processor can execute only one process at a time, from time to time it has to change the currently executed process. Consequently, there are usually processes that are inactive for a certain time interval. The scheduler may change the currently executed process in any one of three different cases, whichever comes first.

- if the process has to wait on a communication point
- if the time slice of the process expired
- if **deschedule_process/0** predicate is called directly by the active process

The first case means that the execution may be suspended because of the synchronization need of the process in some communication handling built-in predicate. In this case, the process becomes suspended and it waits until the appropriate condition fulfills. When next time it gets back the control it continues the execution as if nothing had happened. If the desired condition is already true at the moment when the process reaches the communication point then the process change does not take place. In other words, the scheduler does not take away the processor from the process until it can run continuously.

The second case means that the process may be suspended because it has exhausted its time slice. The time slice for every process is 2 seconds by default. By this, the scheduler avoids that never or rarely communicating processes should monopolize the processor. The size of the time slice can be changed by setting a Prolog flag (see chapter 25).

The third case means that the active process voluntarily yielded the remaining part of its time slice.

8.3 System-wide common names

The scheduler ensures that the system-wide unique names of virtual processors, processes, channels and events are common for the whole system and the consistency of these common names is checked.

8.4 Communication

The scheduler supervises the inter-process communication which is covered by the following three areas:

- channel handling
- message passing
- event passing

The scheduler hides the fact that the communication takes place between different physical processors or inside one processor, so at user level there is no difference.

8.4.1 The channel handling

The scheduler is responsible for the consistent channel handling. Channels do not have to be declared at all, they are created when first time they are referred by one of the **open_channel_for_send/[1,2]** or **open_channel_for_receive/[1,2]** built-in predicates. A channel becomes ready to pass messages when both ends of it are opened. Note that channels can connect only different processes, so no process can hold both ends of a channel.

Channels disappear definitely from the CS-Prolog system when both end of them are released by the **close_channel/1** built-in predicate.

The scheduler keeps track of the actual state of the channels and refuses illegal channel opening and closing requests. However, in a special case the scheduler accepts the opening request even on a busy channel. This can be requested by the 'schedule' argument of the **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** built-in predicates. If this kind of opening is used and if the requested channel is busy (it is opened by someone else), then the scheduler suspends the execution of the caller process until the channel will be free for opening (the former owner closes it by the **close_channel/1** built-in predicate). This technique makes possible to wait for the availability of a channel.

8.4.2 The message transfer

The message passing must take place on channels opened successfully on both, the sending and receiving ends. The message passing is synchronized, which means that both the sender and receiver process must be ready to perform the operation. If either of them is not yet ready, then the other becomes suspended and waits until its partner will be ready too (rendezvous method).

8.4.3 The event passing

The event passing is not a mutual operation as the message passing is. The receiver of an event can be only a real-time process, which is triggered for the event. The sender of an event may be either a process (including real time processes) using the **generate_event/[1,2]** built-in predicate, or the environment of the CS-Prolog system using the function of the similar name of the foreign language interface, or the scheduler itself using its internal clock.

The scheduler executes a real time process the following way. When the real-time process is delegated to a processor, it creates its own event queue. This is a first-in-first-out structure. The incoming events are put in the event queue in the order of their arrival. All events have the same and equal priority. When the real-time process gets the control, first it starts its initialization goal. During the execution of the initialization goal, the real-time process acts as if it were a normal process, but it is already able to accept the incoming events. If the initialization goal has been terminated successfully, then the real time process starts its cyclic behavior. This means that the real time process waits for an event on its event queue. If there is one, the process executes its event-handling goal once. As far as there are events in the queue, the real time process repeats its event-handling goal for each of them. If the queue is empty, the real time process waits until the arrival of the next event.

Obviously, the event-handling goal should be written in such a way that it can act properly for all incoming events. The **get_event/[1,2]** built-in predicate serves for accessing the name and data of the event, which caused the actual execution of the event-handling goal. The **get_event/[1,2]** built-in predicate does not remove the event from the event queue, so it can be called several times. The event will be removed by the scheduler when the actual execution of the event-handling goal is terminated.

The scheduler maintains this cyclic behavior forever, unless the event-handling goal of the real time process fails. If the goal fails, the real time process disappears from the CS-Prolog system, as a normal process would do after the termination of its goal.

Events can be sent from inside a process using the **generate_event/[1,2]** built-in predicate. The event passing is asynchronous, which means that the sender process of the event will never be suspended, the event generating operation always can be performed without any condition.

Generating a specific event kind is not confined to a single process - any number of processes can generate the same kind. The recipient of each event kind, however, must be uniquely defined by the **new_rt/[3,5]** predicates creating the particular real-time processes.

A rudimentary flow control scheme is introduced at system level to prevent a runaway event generator from exhausting memory resources by flooding the system with unconsummated events. For details see: **generate_event/[1,2]**, **set_event_qsize_limit/[1,2]**, **cause_interrupt/2**, and the Prolog flag **discard_mttp**.

Two **system_error** exceptions are defined to cope with asynchronous communication overruns. The **error information** terms are the following:

```
[Goal, 0, event_queue_overrun, EventName]
[Goal, 0, interrupt_queue_overrun, ProcessName]
```

They are routed to the main process, if it is already in terminated state, the system stops.

8.5 The deadlock detection

The scheduler keeps track of the current state of all processes in the program. Among other aims, this enables it to discover global deadlock situations. Global deadlock happens when every process in the CS-Prolog system is suspended for some reason (they are waiting on communication points) so that none of them can be selected for continuing execution. Since this kind of situation could never be resolved, the scheduler signals a run-time error. Of course, the error handling techniques of the CS-Prolog make possible that the user could recover this situation at user level.

The **system_error** exception with the following **error information** term

```
[Goal, 0, deadlock]
```

is routed to the main process. If the main process is already in terminated state, then the system gets stopped with a corresponding error message.

Note that if there are real-time processes in the system which react to timer events (i.e. 'period(T)' or 'idle(T)' had been specified for them as the 5th argument of the corresponding **new_rt/[5,6]**, or any external agent is expected to generate events/interrupts (except CTRL-C from the user), then the momentary lack of activity is not considered a deadlock situation.

8.6 Process deletion and program termination

The scheduler removes the terminated processes from the CS-Prolog system. When a process is removed then the scheduler automatically closes all non-closed channels. When the last process has been terminated the scheduler terminates the CS-Prolog system and returns to the host operating system.

9. Other real-time features

9.1 Time-outs

The CS-Prolog system provides an alarm clock feature. Every process may possess one alarm clock. It can set the alarm clock for a time interval. When the time interval passed the scheduler signals a time-out interrupt at the caller process and then interrupt can be handled by the general interrupt handling techniques. The process can also reset the alarm clock before the exhaustion of the time interval set. This feature is implemented by the **set_timeout/1** and **reset_timeout/0** built-in predicates. The alarm clock is ticking only when the process is executed, so it does not show the real time, rather the time that passed for the process.

9.2 Direct interrupts

The CS-Prolog scheduler provides a possibility to raise an interrupt explicitly for a process (even for itself). The target process will get the interrupt immediately, even if it is suspended for some reason. The interrupt breaks the execution of the suspended call. Then the interrupt can be handled by the general interrupt handling techniques. This feature is implemented by the **cause_interrupt/2** built-in predicate.

Real time tasks cannot be interrupted (at least in any sensible way), because part of the time is spent in the outer loop, where no handler can be installed. Anyway, they can be alerted by events.

PART III

Built-in predicates

10. Introduction

This part contains the full description of the built-in predicate set of CS-Prolog. Built-in predicates are those procedures that are automatically present in any CS-Prolog module.

The argument(s) of a built-in predicate in general cannot be arbitrary Prolog terms; each predicate specifies the valid type and eventually the valid domain of its argument(s). If the argument(s) do not fulfill the requirements, an exception is signaled.

In the description, we used the classification, structure and the wording of the Prolog standard draft.

11. Format of description

Each definition of a built-in predicate consists of four parts: Description, Template and modes, Examples, and Errors.

Description

This part explains the behavior of the predicate. Lists the conditions for which the predicate succeeds, and describes any side effects that occur when it is executed.

Template and modes

This part provides a scheme, which specifies the type and instantiation requirements for each argument of the built-in predicate.

The type of each argument is defined by an identifier, which denotes this argument. The meaning of this identifier is self—evident. The mode of each argument defines whether or not an argument has to be instantiated when the built-in predicate is executed. The mode is one of the following atoms:

+
: the argument has to be instantiated (input)

?
: the argument can be instantiated or can be a variable (any)

@
: the argument will not get instantiated by direct unification within the predicate

-
: the argument has to be a variable that will be instantiated if the predicate succeeds (output).

When appropriate, a **Template and modes** part includes a note that the predicate is a predefined operator.

Examples for a template:

`arg(+integer, +compound_term, ?term)`
: The first argument has to be an integer number, the second argument has to be a compound term, and the third argument can be any term (including variable).

`open(@source_sink, @io_mode, -stream)`
: The first argument has to be a name of a source or a sink, the second has to be an atom denoting an I/O mode, and the third argument has to be an unbound variable. The third argument will be unified with a stream identifier, if the predicate succeeds. The first two arguments will not be changed

Examples

An example is normally the built-in predicate used as a goal, together with a statement saying whether the goal succeeds, fails or produces an error. The statement also describes any side effect and substitution that occurs.

Errors

A list of the circumstances that will cause an error when the built-in predicate is executed, together with the sort of error that is caused.

The explanation of the error will always refer to the call of the predicate as it is shown in the **Description** part. If the **Argno** or **Other** element of the error info term contain relevant information, they are mentioned as **ErrInfo-Argno** and **ErrInfo-Other**.

ISO Compliance

CS-Prolog implements all built-in predicates listed in International Standard ISO/IEC 13211-1 (Prolog - Part 1: General core). CS-Prolog does not offer a strictly conforming mode, which would reject uses of built-in predicates that are specific to CS-Prolog. To help programmers who want to write standard compliant Prolog programs the standard predicates are annotated with **`[ISO]`** in the *Template and modes* part.

12. Term unification

These predicates deal with the unification of two terms.

(=) /2 - Prolog unify

Description

`X = Y`

The predicate unifies **X** and **Y** and succeeds if they are unifiable, otherwise it fails. If during the unification a variable is bound to a compound term that contains the same variable, the unification succeeds, creating a cyclic term. (There is no occurs check during the unification.) Such a cyclic term can cause an error or an endless loop when unifying it with another term or when writing out. See also **unify_with_occurs_check/2** predicate.

Template and modes

`?term = ?term` [ISO]

= is a predefined infix operator.

Examples

```
1 = 1.
```
Succeeds.
```
f(X, aa) = f(aa, Y)
```
Succeeds, unifying the two variables **X** and **Y** with **aa**.
```
1 = 2.
```
Fails.
```
X = a(X).
```
Succeeds, creating a dangerous cyclic term.

Errors

None

unify_with_occurs_check/2 - safe unify

Description

`unify_with_occurs_check(X, Y)`

Performs unify with occurs check. That means that if **X** and **Y** are not unifiable or **X** and **Y** are unifiable and no cyclic term is created during unification, then this predicate has the same effect as the previous one, **=/2**. If during the unification of **X** and **Y** a cyclic term would be created, the predicate fails.

Template and modes

`unify_with_occurs_check(?term, ?term)` [ISO]

Examples

```
unify_with_occurs_check(1, 1).
```
Succeeds.
```
unify_with_occurs_check(f(X, aa), f(aa, Y)).
```
Succeeds, unifying the two variables **X** and **Y** with **aa**.
```
unify_with_occurs_check(1, 2).
```
Fails.

```
unify_with_occurs_check(X, a(X)).
```
Fails.

Errors

None

(\=) /2 - not Prolog unifiable

Description

```
X \= Y
```

Succeeds if and only if **X** and **Y** are not unifiable.

Template and modes

```
@term \= @term                                                    [ISO]
```

\= is a predefined infix operator

Examples

```
1 \= 1.
```
Fails.
```
f(X, aa) \= f(aa, Y)
```
Fails.
```
1 \= 2.
```
Succeeds
```
X \= a(X).
```
Fails.

Errors

None

13. Type testing

These predicates test the type associated with a term. Each of these predicates simply succeeds or fails; there is no side effect, substitution, or error.

According to the Prolog Standard, every term has one of the following mutually-exclusive types: variable, integer, floating point value, atom, compound term.

The CLP extension to CS-Prolog adds a sixth type to this categorization: constrained variable. Terms of this type appear only dynamically, when the program actively uses CLP; there is no syntactic construct to represent a constarined variable in the source program. Constrained variables can be be bound transparently to (some) numbers.

> NOTE: Prolog is not a typed language, and an argument of a compound term or a predicate can be any term whatsoever. Although the control constructs, built-in predicates and evaluable functors are defined for all arguments and operands, it is often an error if an argument does not have a particular sort of value.
>
> It is therefore convenient to classify the terms as belonging to one of several disjoint types.

var/1

Description

`var(X)`

Is true if and only if **X** is a variable.

Template and modes

`var(@term)` [ISO]

Examples

`var(foo).`
Fails.

`var(Foo).`
Succeeds.

`foo=Foo, var(Foo).`
Fails.

`var(_).`
Succeeds.

Errors

None

atom/1

Description

`atom(X)`

Is true if and only if **X** is an atom.

Template and modes

`atom(@term)` [ISO]

Examples

```
atom(foo).
```
Succeeds.

```
atom(Foo).
```
Fails.

```
atom('Foo').
```
Succeeds.

```
atom(foo(Foo)).
```
Fails.

```
atom([]).
```
Succeeds.

```
atom(6).
```
Fails.

```
atom(3.3).
```
Fails.

Errors

None

integer/1

Description

```
integer(X)
```

Is true if and only if **X** is an integer.

Template and modes

```
integer(@term)                                                    [ISO]
```

Examples

```
integer(6).
```
Succeeds.

```
integer(foo).
```
Fails.

```
integer(Foo).
```
Fails.

```
integer(foo(Foo)).
```
Fails.

```
integer(3.3).
```
Fails.

Errors

None

float/1

real/1

Description

```
real(X)
```

Is equivalent with

```
float(X)
```

`float(X)`

Is true if and only if **X** is a floating-point number.

Template and modes

`float(@term)` [ISO]

`real(@term)`

Examples

```
real(3.3).
```
Succeeds.
```
real(6).
```
Fails.
```
real(foo).
```
Fails.
```
real(Foo).
```
Fails.
```
real(foo(Foo)).
```
Fails.

Errors

None

atomic/1

Description

`atomic(X)`

Is true if and only if **X** is an atom, an integer, or a floating-point number.

Template and modes

`atomic(@term)` [ISO]

Examples

```
atomic(foo).
```
Succeeds.
```
atomic(Foo).
```
Fails.
```
atomic(foo(Foo)).
```
Fails.
```
atomic([]).
```
Succeeds.
```
atomic(6).
```
Succeeds.
```
atomic(3.3).
```
Succeeds.

Errors

None

compound/1

Description

`compound(X)`

Is true if and only if **X** is a compound term.

Template and modes

`compound(@term)` [ISO]

Examples

`compound(foo).`
Fails.
`compound(Foo).`
Fails.
`compound(foo(Foo)).`
Succeeds.
`compound([]).`
Fails.
`compound([6]).`
Succeeds.
`compound("1234").`
Succeeds.
`compound(3.3).`
Fails.

Errors

None

nonvar/1

Description

`nonvar(X)`

Is true if and only if **X** is not a variable (see also **strict_nonvar/1**).

Template and modes

`nonvar(@term)` [ISO]

Examples

`nonvar(foo).`
Succeeds.
`nonvar(Foo).`
Fails if **Foo** is currently unbound, otherwise succeeds.
`foo=Foo, nonvar(Foo).`
Succeeds.
`nonvar(_).`
Fails.

Errors

None

number/1

Description

`number(X)`

Is true if and only if **X** is an integer or a floating-point number.

Template and modes

`number(@term)` `[ISO]`

Examples

`number(6).`
Succeeds.
`number(foo).`
Fails.
`number(_).`
Fails.
`number(foo(Foo)).`
Fails.
`number(3.3).`
Succeeds.

Errors

None

ground/1

Description

`ground(X)`

Is true if and only if **X** is completely instantiated, i.e., free of unbound variables (see also **strict_ground/1**).

Template and modes

`ground(@term)`

Examples

`ground(foo).`
Succeeds.
`ground(Foo).`
Fails if **Foo** is currently unbound or is bound to a term containing any unbound variable as subterm, otherwise succeeds.
`ground(foo(_)).`
Fails.
`ground([1, 2.0, foo(bar)]).`
Succeeds.
`ground([a | _]).`
Fails.

Errors

None

constrained_var/1

Description

`constrained_var(X)`

Is true if and only if **X** is bound to a constrained variable (an object representing a numeric value with constraints imposed on it, managed by a CLP solver). This is possible only if the CS-Prolog runtime program is configured with at least one solver, and the Prolog program being executed actively uses the CLP feature.

Note that a constrained variable itself can later be bound to a number, which renders it 'transparent'.

Note also that there is no syntactic construct to represent a constrained variable in the source program.

Template and modes

`constrained_var(@term)` [ISO]

Examples

`constrained_var(foo).`
Fails.

`constrained_var(Foo).`
Succeeds if **Foo** is currently bound to a constrained variable, otherwise fails. (Primary reference to a constrained variable is obtained when an unbound variable is passed to a CLP solver within a constraint in a successful call of e.g. **clp_constraint/1**.)

`clp_constraint([Foo =< 10]), constrained_var(Foo).`
Succeeds if the first call succeeds.

`constrained_var(_).`
Fails.

Errors

None

strict_nonvar/1

Description

`strict_nonvar(X)`

Is true if and only if **X** is neither an unbound variable, nor a constrained variable. If there is no active CLP solver in the CS-Prolog process where the call is issued then this call is equivalent with **`nonvar(X)`**.

Template and modes

`strict_nonvar(@term)` [ISO]

Examples

`strict_nonvar(foo).`
Succeeds.

`strict_nonvar(Foo).`
Fails if **Foo** currently either is unbound or is bound to a constrained variable, otherwise succeeds.

`strict_nonvar(_).`
Fails.

Errors

None

strict_ground/1

Description

`strict_ground(X)`

Is true if and only if **X** is completely and strictly instantiated, i.e., free of unbound variables and constrained variables. If there is no active CLP solver in the CS-Prolog process where the call is issued then this call is equivalent with **`ground(X)`**.

Template and modes

`strict_ground(@term)`

Errors

None

14. Term comparison

These predicates test the ordering of two terms. Each of these predicates simply succeeds or fails; there is no side effect, substitution, or error.

14.1 Term order

There is an ordering **term_precedes** defined for any two terms.

If **X** and **Y** are identical terms then **X term_precedes Y** and **Y term_precedes X** are both false.

If **X** and **Y** have different types, **X term_precedes Y** if and only if the type of **X** precedes the type of **Y** in the following order:

- variable
- constrained variable (only with CLP extension)
- floating-point number
- integer
- atom (symbol)
- compound

If **X** and **Y** are variables which are not identical then the term order of these terms is defined by their internal representation.

If **X** and **Y** are floating point numbers then **X term_precedes Y** if and only if **X** < **Y**.

If **X** and **Y** are integers then **X term_precedes Y** if and only if **X** < **Y**.

If **X** and **Y** are atoms then **X term_precedes Y** if and only if **X** is less then **Y** in lexicographic ordering.

If **X** and **Y** are compound terms then **X term_precedes Y** if and only if:

- The arity of **X** is less than the arity of **Y** or

- **X** and **Y** have the same arity, and the functor name of **X** is **FX**, and the functor name of **Y** is **FY**, and **FX term_precedes FY** or

- **X** and **Y** have the same functor name and arity, and there is a positive integer **N** such that, for all **I** less than **N** the **I**th argument of **X** is identical with the **I**th argument of **Y**, and the **N**th argument of **X term_precedes** the **N**th argument of **Y**.

(==) /2 - identical,

(\==) /2 - not identical

Description

`X == Y`

Is true if and only if **X** and **Y** are identical terms.

`X \== Y`

Is true if and only if **X** and **Y** are not identical terms.

Template and modes

```
@term == @term                                    [ISO]
@term \== @term                                   [ISO]
```

== and \== are predefined infix operators.

Examples

```
1 == 1.
```
Succeeds.

```
X == X.
```
Succeeds.
```
1 == 2.
```
Fails.
```
X == 1.
```
Fails.
```
X == Y.
```
Fails.
```
1 \== 2.
```
Succeeds.
```
X \== 1.
```
Succeeds.

Errors

None

(@<) /2 - term less than,

(@=<) /2 - term less than or equal

Description

```
X @< Y
```

Is true if and only if **X term_precedes Y**.

```
X @=< Y
```

Is true if and only if **X term_precedes Y** or **X** and **Y** are identical terms.

Template and modes

```
@term @< @term                                                    [ISO]
@term @=< @term                                                   [ISO]
```

@< and **@=<** are predefined infix operators.

Examples

```
1.0 @< 1.
```
Succeeds.
```
cat @< dog.
```
Succeeds.
```
short @< short.
```
Fails.
```
short @< shorter.
```
Succeeds.
```
zoo @< apple(pie).
```
Succeeds.
```
foo(b) @< foo(a).
```
Fails.
```
foo(a,b) @< yupp(a).
```
Fails.
```
X @< X.
```
Fails.
```
1.0 @=< 1.
```
Succeeds.
```
short @=< short.
```
Succeeds.

Errors

None

(@>) /2 - term greater than,

(@>=) /2 - term greater than or equal

Description

```
X @> Y
```

Is true if and only if **Y term_precedes X**.

```
X @>= Y
```

Is true if and only if **Y term_precedes X** or **X** and **Y** are identical terms.

Template and modes

```
@term @> @term                                                    [ISO]
@term @>= @term                                                   [ISO]
```

@> and **@>=** are predefined infix operators.

Examples

```
1.0 @> 1.
```
Fails.

```
cat @> dog.
```
Fails.

```
short @> short.
```
Fails.

```
short @> shorter.
```
Fails.

```
zoo @> apple(pie).
```
Fails.

```
foo(b) @> foo(a).
```
Succeeds.

```
foo(a,b) @> yupp(a).
```
Succeeds.

```
X @> X.
```
Fails.

```
1.0 @>= 1.
```
Fails.

```
short @>= short.
```
Succeeds.

Errors

None

15. Term creation and decomposition

These predicates enable a term to be assembled from its component parts, or split into its component parts, or copied.

functor/3

Description

`functor(Term, Name, Arity)`

If **Term** is a constant then **Name** is unified with **Term**, and **Arity** is unified with 0.

If **Term** is a compound term then **Name** is unified with the name of **Term**, and **Arity** is unified with the arity of the Term.

If **Term** is a variable, **Name** is a constant, and **Arity** is 0 then **Term** is unified with **Name**.

If **Term** is a variable, **Name** is an atom, and **Arity** is an integer greater than zero then **Term** is unified with a term that has name **Name** and arity **Arity**, and **Arity** distinct uninstantiated arguments.

Template and modes

```
functor(@nonvar, ?constant, ?integer)                    [ISO]
functor(-var, +constant, +integer)                       [ISO]
```

Examples

`functor(foo(a, b, c), foo, 3).`
Succeeds.

`functor(foo(a, b, c), X, Y).`
Succeeds, unifying **`X`** with **`foo`** and **`Y`** with **`3`**.

`functor(X, foo, 3).`
Succeeds, unifying **`X`** with **`foo(_, _, _)`**.

`functor(X, foo, 0).`
Succeeds, unifying **`X`** with **`foo`**.

`functor(foo(a), foo, 2).`
Fails.

`functor(1, X, Y).`
Succeeds, unifying **`X`** with **`1`** and **`Y`** with **`0`**.

`functor(X, 1.1, 0).`
Succeeds, unifying **`X`** with **`1.1`**.

`functor([_|_], '.', 2).`
Succeeds.

`functor([], [], 0).`
succeeds.

Errors

`instantiation_error`
Term is uninstantiated and at least one of **Name** and **Arity** is uninstantiated.

`type_error(atomic, Name)`
Name is not a variable and not atomic

`type_error(atom, Name)`
Arity is a positive integer and **Name** is atomic but not an atom

`type_error(integer, Arity)`
Arity is not a variable and not an integer

`domain_error(not_less_than_zero, Arity)`
Arity is an integer that is less than zero

`representation_error(max_arity)`
Term is uninstantiated and **Arity** is greater than 255.

arg/3

Description

`arg(N, Term, Arg)`

Unifies **Arg** with the **N**th argument of the compound term **Term**. Arguments are numbered from 1.

Template and modes

`arg(+integer, +compound_term, ?term)` [ISO]

Examples

`arg(1, foo(a, b), a).`
Succeeds.
`arg(1, foo(a, b), X).`
Succeeds, unifying **X** with **a**.
`arg(1, foo(X, b), a).`
Succeeds, unifying **X** with **a**.
`arg(0, foo(X, b), foo).`
Fails.
`arg(1, foo(X, b), Y).`
Succeeds, unifying **X** with **Y**.
`arg(1, foo(a, b), b).`
Fails.

Errors

`instantiation_error`
N is uninstantiated.
`instantiation_error`
Term is uninstantiated.
`type_error(integer, N)`
N is not a variable and not an integer.
`type_error(compound, Term)`
Term is not a variable and not a compound term.
`domain_error(not_less_than_zero, N)`
N is an integer that is less than zero.

(=..) /2 - univ

Description

`Term =.. List`

If **Term** is a constant then **List** is unified with the list whose only element is **Term**.

If **Term** is a compound term then **List** is unified with the list whose **head** is the name of **Term** and whose **tail** is the list of its arguments.

If **Term** is a variable and **List** is a list whose only element is a constant then **Term** is unified with the single element of **List**.

If **Term** is a variable and **List** is a list and there exists a compound term **CT** such that the name of **CT** is the **head** of **List** and the list of the arguments of **CT** is the **tail** of **List** then **CT** is created and **Term** is unified with it.

Template and modes

```
+nonvar =.. ?list                                    [ISO]
-var =.. +list                                       [ISO]
```

=.. is a predefined infix operator

Examples

`foo(a, b)=.. [foo, a, b].`
Succeeds.

`X =.. [foo, a, b].`
Succeeds, unifying **X** with **foo(a, b)**.

`foo(a, b) =.. L.`
Succeeds, unifying **L** with **[foo, a, b]**.

`foo(X, b) =.. [foo, a, Y].`
Succeeds, unifying **X** with **a** and **Y** with **b**.

`1 =.. [1].`
Succeeds.

Errors

`instantiation_error`
Term is uninstantiated and **List** is a partial list.

`instantiation_error`
Term is uninstantiated and **List** is a list whose head is uninstantiated.

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

`type_error(atomic, Head)`
List is of the form [**Head**] and **Head** is a compound term.

`type_error(atom, Head)`
List is of the form [**Head | Tail**], **Head** is neither an atom nor a variable and **Tail** is not the empty list.

`domain_error(non_empty_list, List)`
Term is a variable and List is the empty list.

`representation_error(Term =.. List, 2, max_arity, [])`
The length of **List** exceeds the maximum value allowed for an arity.

copy_term/2

Description

`copy_term(Term_1, Term_2)`

Copies **Term_1** to a term **T** while replacing each variable of **Term_1** by a new variable. If **Term_2** can be unified with **T**, the predicate succeeds. Otherwise, the predicate fails.

Template and modes

```
copy_term(@term, ?term)                              [ISO]
```

Examples

`copy_term(X, Y).`
Succeeds. **X** and **Y** remain distinct variables.

`copy_term(X, 3).`
Succeeds. **X** remains a variable.

`copy_term(_, a).`
Succeeds.

`copy_term(a+X, X+b).`
Succeeds, unifying **X** with **a**. This example contradicts to the mode of the first argument (@ that is not modified). It is not caused by the **copy_term/2**, but it is because the two arguments contain the same variable.

`copy_term(X+X+Y, A+B+B).`
Succeeds, unifying **A** with **B**. **X** and **Y** remain distinct variables.

`copy_term(a, b).`
Fails.

`copy_term(a+X, X+b), copy_term(a+X, X+b).`
Fails.

Errors

`representation_error(max_term)`
The term to be copied (first argument) is larger then the limit set for built-in predicates (the size of terms created in one step is restricted)

numbervars/3

Description

`numbervars(Term,N,M)`

Unifies each of the variables in term **Term** with a special term '**$VAR(I)**' where **I** ranges from **N** to **NN**. These terms are printed out by a

```
write(Term,[numbervars(true)])
```

call as a capital letter followed by an integer (see section 3.11 for detailed description). **M** is unified with the value of the expression **NN + 1**.

Template and modes

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

Examples

`numbervars(a(X,Y),3,M)`
X is unified with **'$VAR(3)'**, **Y** is unified with **'$VAR(4)'**, **M** is unified with 5. If this term is written out with **numbervars(true)** option, the output is

```
a(D,E)
```

Errors

`instantiation_error`
N is uninstantiated.

`type_error(integer, N)`
N is not an integer.

`domain_error(not_less_than_zero, N)`
N is negative.

16. Arithmetic evaluation

These predicates evaluate an arithmetic expression and unify the result with a term, or compare results of evaluation of two arithmetic expressions.

Integer arithmetic is performed internally with a greater precision than allowed by the CSP-II integer range, and only the final result is checked. Therefore, it may occur that a complex expression is evaluated successfully, while some subexpression from it causes integer overflow if evaluated separately.

Floating-point operations are influenced by the current value of the **`float_range_checking_function`** prolog flag. The setting influences the action taken when an operation yields a denormalized value. If the current setting is **`denormalize`**, then the result is accepted. When the setting is **`underflow_to_zero_after_rounding`** then the result is forced to 0.0. With the last possibility, **`underflow_exception_after_rounding`**, an exception is raised.

16.1 Arithmetic expressions

A Prolog term is an arithmetic expression if it is composed from numbers and evaluable functors. Each evaluable functor corresponds to an arithmetic operation on the argument(s) of the functor.

The following unary operations are available:

`-`
: Arithmetic negation, for integer argument gives integer result, for float argument gives float result.

`\`
: Bitwise complement, defined only for integer argument, gives integer result.

`abs`
: Absolute value, for integer argument gives integer result, for float argument gives float result.

`sign`
: Sign of the argument (-1, 0, or 1). For integer argument gives integer result, for float argument gives float result.

`float`
: Returns its argument as a float value. Accepts integer argument only and gives float result. (This is not to be confused with the **float/1** test predicate, the ISO equivalent of **real/1**.)

`sqrt`
: Square root of the argument. For any number as argument gives a float result.

`float_significand`
: Float significand (mantissa) of the argument. For any number as argument gives a float result (this result is between -1.0 and 1.0, noninclusive).

`float_exponent`
: Float exponent of the argument. Defined for any type of number, gives integer result.

`exp`
: Exponential function of the argument ($\exp(x)=e^x$). For any number gives float result.

`log`
: Natural logarithm of the argument. For any number gives float result.

`log10`
: Base-10 logarithm of the argument. For any number gives float result.

`sin`
: Sine of the argument. For any number gives float result.

`cos`
: Cosine of the argument. For any number gives float result.

`tan`
: Tangent of the argument. For any number gives float result.

`asin`
: Arc sine of the argument. For any number gives float result.

`acos`
: Arc cosine of the argument. For any number gives float result.

`atan`
: Arc tangent of the argument. For any number gives float result.

`float_successor`
: Float successor of the argument. For any number gives float result. It is the next representable float number that is larger than the argument.

`float_predecessor`
: Float predecessor of the argument. For any number gives float result. It is the previous representable float number that is smaller than the argument.

`float_unit_in_last_place`
: Float unit in last place of the argument. For any number gives float result.

`float_integer_part`
: Float integer part of the argument. For any number gives float result.

`float_fractional_part`
: Float fractional part of the argument. For any number gives float result.

`floor`
: The largest integer not greater then the argument. For any number, the result is an integer.

`ceiling`
: The smallest integer not less then the argument. For any number, the result is an integer.

`truncate`
: An integer created by truncating the fractional part of the argument. For any number, the result is an integer. (This function is equivalent with **floor** for positive numbers and it is equivalent with **ceiling** for negative numbers.)

`round`
: The integer nearest to the argument. For any number, the result is an integer. (This operation is defined as follows:

```
round(X) = floor(X + 0.5)
```

The following binary operations are available:

`+`
: Addition of arguments. Defined for all arithmetic types. If both arguments are integers the result is an integer, otherwise the result is a float number.

`-`
: Subtraction of arguments. Defined for all arithmetic types. If both arguments are integers the result is an integer, otherwise the result is a float number.

`*`
: Multiplication of arguments. Defined for all arithmetic types. If both arguments are integers the result is an integer, otherwise the result is a float number.

`/`
: Division of first argument by the second. Defined for all arithmetic types. Always gives float result, even if the division of two integer arguments would be an integer number.

`mod`
: Gives the remainder when dividing the first argument by the second. Its sign is identical with the sign of the divisor. Defined for integer arguments. Gives integer result.

`rem`
: Gives the remainder when dividing the first argument by the second. Its sign is identical with the sign of the dividend. Defined for integer arguments. Gives integer result.

`**`
: Raising to power operation. Defined for both integer and float arguments. The result is always a float number. Zero raised to power zero gives 1.0.

`//`
: Integer division of arguments. Defined for integer arguments. Gives integer result. Rounding is towards zero.

`>>`
: Bitwise logical right shift of arguments. Defined for integer arguments. Gives integer result. The shift works on 2's complement representation of CS-Prolog integers as **unsigned** values. The result is a signed value, the sign depending on the MSB. Negative second argument reverses the direction of the shift operation.

`<<`
: Bitwise logical left shift of arguments. Defined for integer arguments. Gives integer result. The shift works on 2's complement representation of CS-Prolog integers as **unsigned** values. The result is a signed value, the sign depending on the MSB. Negative second argument reverses the direction of the shift operation.

`/\`
: Bitwise and of arguments. Defined for integer arguments. Gives signed integer result.

`\/`
: Bitwise or of arguments. Defined for integer arguments. Gives signed integer result.

`hypot`
: Euclidean distance function of arguments. Defined for all arithmetic types, gives always float result. The definition of this function is:

```
hypot(X,Y) = sqrt(X*X + Y*Y)
```

`min`
: Minimum value of arguments. Defined for all arithmetic types. If both arguments are integers the result is an integer, otherwise the result is a float number.

`max`
: Maximum value of arguments. Defined for all arithmetic types. If both arguments are integers the result is an integer, otherwise the result is a float number.

`float_scale`
: Float scale of arguments. Defined for any arithmetic type as first argument and integers as second argument. Gives float result. The definition of this operation is:

```
float_scale(x,y) = x * 2**y.
```

`float_truncate`
: Float truncate of arguments. Defined for any number as first argument and integers as second argument. Gives float result. It works on the conceptual representation of **float_exponent** and **float_significand** and truncates the significand at the digit indicated by the second argument.

`float_round`
: Float round of arguments. Defined for any number as first argument and integers as second argument. Gives float result. It works on the conceptual representation of **float_exponent** and **float_significand**, and rounds the significand at the digit indicated by the second argument.

(is)/2 - evaluate expression

Description

`Result is Expression`

Evaluates the arithmetic expression **Expression** and unifies **Result** with the resulting value.

Template and modes

`?term is +arithmetic_expr` [ISO]

is is a predefined infix operator.

Examples

```
X is 7 * (30 + 5).
```
Succeeds, unifying **X** with **245**.

```
X is 35 / 7.
```
Succeeds, unifying **X** is with **5.0**.

```
X is 35 // 7.
```
Succeeds, unifying **X** is with **5**.
```
X is floor(7.6).
```
Succeeds, unifying **X** is with **7**.
```
X is round(7.6).
```
Succeeds, unifying **X** is with **8**.
```
X is float(7).
```
Succeeds, unifying **X** is with **7.0**.
```
X is 2 ** 4 >> 2.
```
Succeeds, unifying **X** is with **4**.
```
X is 2 ** 4 << 2.
```
Succeeds, unifying **X** is with **64**.
```
X is 10 /\ 12
```
Succeeds, unifying **X** is with **8**.
```
X is 10 \/ 12
```
Succeeds, unifying **X** is with **14**

Errors

```
instantiation_error
```
Expression is insufficiently instantiated.
```
type_error(integer, Culprit)
```
Evaluable functor in **Expression** requires integer value. **Culprit** is not an integer.
```
type_error(evaluable, CulpritFunctor)
```
There is a subexpression of **Expression** that is an atom or a compound term, and its functor **CulpritFunctor** is not an evaluable functor.
```
evaluation_error(Flag)
```
An error indicated by **Flag** occurred while evaluating **Expression**. **Flag** is one of the following flags: **underflow**, **int_overflow**, **float_overflow**, **undefined** and **zero_divisor**.

(=:=) /2 - arithmetic equal

(=\=) /2 - arithmetic not equal

Description

```
E1 =:= E2
```
Evaluates the expressions **E1** and **E2**, and succeeds if the results are equal numbers, otherwise fails.
```
E1 =\= E2
```
Evaluates the expressions **E1** and **E2**, and fails if the results are equal numbers, otherwise succeeds.

Template and modes

```
+arithmetic_expr =:= +arithmetic_expr                    [ISO]
+arithmetic_expr =\= +arithmetic_expr                    [ISO]
```
=:= and =\= are predefined infix operators.

Examples

```
5 + 6 =:= 22 // 2.
```
Succeeds
```
round(5.6) =:= 5.
```
Fails
```
5 + 6 =\= 22 // 2.
```
Fails

```
round(5.6) =\= 5.
```
Succeeds

Errors

`instantiation_error`
E1 or **E2** is insufficiently instantiated.

`type_error(integer, Culprit)`
Evaluable functor in **E1** or **E2** requires integer value. **Culprit** is not an integer.

`type_error(evaluable, CulpritFunctor)`
There is a subexpression of **E1** or **E2** that is an atom or a compound term, and its functor **CulpritFunctor** is not an evaluable functor.

`evaluation_error(Flag)`
An error indicated by **Flag** occurred while evaluating **E1** or **E2**. **Flag** is one of the following flags: **underflow**, **int_overflow**, **float_overflow**, **undefined** and **zero_divisor**.

(<) /2 - arithmetic less

(=<) /2 - arithmetic less or equal

Description

`E1 < E2`

Evaluates the expressions **E1** and **E2** and succeeds if the result of evaluation of **E1** is less than the result of evaluation of **E2**, otherwise fails.

`E1 =< E2`

Evaluates the expressions **E1** and **E2** and succeeds if the result of evaluation of **E1** is less than or equal to the result of evaluation of **E2**, otherwise fails.

Template and modes

```
+arithmetic_expr < +arithmetic_expr                    [ISO]
+arithmetic_expr =< +arithmetic_expr                   [ISO]
```

< and =< are predefined infix operators.

Examples

```
5 + 6 =< 22 // 2.
```
Succeeds

```
5 < round(5.6).
```
Succeeds

Errors

`instantiation_error`
E1 or **E2** is insufficiently instantiated.

`type_error(integer, Culprit)`
Evaluable functor in **E1** or **E2** requires integer value. **Culprit** is not an integer.

`type_error(evaluable, CulpritFunctor)`
There is a subexpression of **E1** or **E2** that is an atom or a compound term, and its functor **CulpritFunctor** is not an evaluable functor.

`evaluation_error(Flag)`
An error indicated by **Flag** occurred while evaluating **E1** or **E2**. **Flag** is one of the following flags: **underflow**, **int_overflow**, **float_overflow**, **undefined** and **zero_divisor**.

(>) /2 - arithmetic greater

(>=) /2 - arithmetic greater or equal

Description

`E1 > E2`

Evaluates the expressions **E1** and **E2** and succeeds if the result of evaluation of **E1** is greater than the result of evaluation of **E2**, otherwise fails.

`E1 >= E2`

Evaluates the expressions **E1** and **E2** and succeeds if the result of evaluation of **E1** is greater than or equal to the result of evaluation of **E2**, otherwise fails.

Template and modes

```
+arithmetic_expr > +arithmetic_expr                    [ISO]
+arithmetic_expr >= +arithmetic_expr                   [ISO]
```

> and >= are predefined infix operators.

Examples

```
5 + 6 >= 22 // 2.
```
Succeeds

```
5 > round(5.6).
```
Fails

Errors

`instantiation_error`
E1 or **E2** is insufficiently instantiated.

`type_error(integer, Culprit)`
Evaluable functor in **E1** or **E2** requires integer value. **Culprit** is not an integer.

`type_error(evaluable, CulpritFunctor)`
There is a subexpression of **E1** or **E2** that is an atom or a compound term, and its functor **CulpritFunctor** is not an evaluable functor.

`evaluation_error(Flag)`
An error indicated by **Flag** occurred while evaluating **E1** or **E2**. **Flag** is one of the following flags: **underflow**, **int_overflow**, **float_overflow**, **undefined** and **zero_divisor**.

random/1

Description

`random(X)`

Unifies **X** with the next element of a series of pseudo random integers in the range from 0 to the largest CS-Prolog integer. (The largest integer is the value associated with the Prolog flag **max_integer** and can be obtained using the **get_prolog_flag/2** predicate.)

The **set_random_seed/1** predicate can be used to set a new starting point for the pseudo random series.

The random seed (the starting point of the generator) is local to the process. Initially all processes start with the same number.

Template and modes

`random(-integer)`

Errors

`type_error(variable, X)`
X is not an unbound variable.

set_random_seed/1

Description

`set_random_seed(Seed)`

Sets the new starting point for generating a series of pseudorandom integers. The new starting point depends on the **Seed** argument. If it is negative, a randomly chosen integer will be set (the current time is used to get a genuine random value). If **Seed** is 0, the generator is reset to its initial value. Positive arguments will become themselves the new base value for the generator.

The **random/1** predicate can be used to retrieve the pseudorandom numbers generated.

The random seed is local to the process. The initial value is the same for all processes.

Template and modes

`set_random_seed(+integer)`

Examples

`set_random_seed(-1).`
Sets randomly the base value of the generator.
`set_random_seed(1994).`
Sets the base value of the generator to **1994**.

Errors

`instantiation_error`
Seed is a variable.
`type_error(integer, Seed)`
Seed is not a variable and not an integer.

17. Clause retrieval and information

These predicates enable the contents of the Prolog database to be inspected during execution.

clause/2

Description

`clause(Head, Body)`

This predicate succeeds if the functor of **Head** corresponds to a dynamic predicate, and there is a clause in the database with term form **H :- B** and **H** can be unified with **Head** `and` **B** can be unified with **Body**. Facts are considered as rules having **true** as their body. The clause is searched only in one module, if **Head** contains a module name prefix then in that module, if it doesn't, in the current module.

This predicate is resatisfiable. On backtracking, it gives all possible clauses. If the predicate is modified after a **clause/2** call, the modifications are not effective from the point of view of the **clause/2** call. The predicate behaves as if it was frozen in the moment of the call.

If the functor of **Head** corresponds to a static or built-in procedure, **clause/2** raises an exception. If the functor does not correspond to any procedure, the call simply fails.

Template and modes

`clause(+head, ?body)` `[ISO]`

Examples

`clause(foo, true).`
Succeeds if there is a dynamic fact **`foo`** or a rule **`foo :- true`**.

`clause(mod:x(_), Body).`
Succeeds if there is a dynamic clause for **`x/1`** in module **`mod`** and unifies the body with **`Body`**, or if there is a dynamic fact for **`x/1`** in module **`mod`** and unifies **`true`** with **`Body`**.

Errors

`instantiation_error`
Head or the module prefix extracted from **Head** is uninstantiated.
`type_error(callable, Head)`
Head is neither a variable nor a callable term.
`type_error(callable, Body)`
Body is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod module prefix extracted from **Head** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Head** is not a module name.
`permission_error(access, private_procedure, CulpritFunc)`
CulpritFunc is the principal functor of **Head** and is the functor of a private procedure (built-in or static).

get_clause/3

Description

`get_clause(PredInd,N,Clause)`

Succeeds if **PredInd** is a functor of a dynamic predicate, and the **N**th clause of the predicate is unifiable with **Clause**. If **PredInd** contains a module name prefix then the clause is searched in that module. If **PredInd**

doesn't contain module prefix, the current module is searched. If the number of clauses in the predicate is less than **N** then an exception is raised.

Template and modes

```
get_clause(+predicate_specification, +integer, ?clause)
```

Examples

```
get_clause(mod1:foo/2,3,X)
```
Unifies the third clause of predicate **foo/2** in module **mod1** with **X**.

Errors

`instantiation_error`
PredInd is insufficiently instantiated.
`instantiation_error`
N is a variable
`type_error(integer, N)`
N is not an integer
`domain_error(not_less_than_zero, N)`
N is less than zero.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0.
`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(access,private_procedure, PS)`
PS is the predicate specification and it is the functor of a built-in or static procedure.
`existence_error(procedure, PS)`
PS is the predicate specification and it is not a functor of a dynamic procedure. **ErrInfo-Argno** is 1.
`existence_error(procedure, N)`
There is no **N**th clause in predicate. **ErrInfo-Argno** is 2.

clause_count/2

Description

```
clause_count(PredInd,N)
```

Succeeds if **PredInd** is a functor of a dynamic predicate and **N** is unified with the number of clauses in this predicate. **PredInd** can contain a module name prefix, if it does then the predicate is searched in the specified module, if it doesn't, the current module is searched.

Template and modes

```
clause_count(+predicate_specification, -number)
```

Examples

```
clause_count(mod1:foo/2,N)
```
Unifies **N** with number of clauses of predicate **foo/2** in module **mod1**.

Errors

`instantiation_error`
PredInd is insufficiently instantiated.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is a negative integer.
`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(access,private_procedure, PS)`
PS is the predicate specification and it is the functor of a built-in or static procedure.

current_predicate/1

Description

`current_predicate(PredInd)`

Succeeds if **PredInd** is a functor of one of the user-defined procedures in the database. **PredInd** has to be a variable or a functor expression of form **Name / Arity**, without a module prefix.

This predicate is resatisfiable. On backtracking, it gives all the possible functors. If the database is modified after a **current_predicate/1** call, the modifications will not be effective from the point of view of the **current_predicate/1** call.

Each user-defined procedure is found, whether static or dynamic. A user-defined procedure is also found even when all its clauses have been previously retracted.

This predicate lists the procedure functors of all modules. To find a functor in a specific module use **current_predicate/2**.

Template and modes

`current_predicate(?predicate_indicator)` `[ISO]`

Examples

`current_predicate(foo/2).`
Succeeds if there is a user-defined procedure **`foo/2`**.
`current_predicate(current_predicate/1).`
Fails.
`current_predicate(foo/Arity).`
Succeeds unifying **`Arity`** with the arities of all user-defined procedures called **`foo`**.

Errors

`type_error(predicate_indicator, PredInd)`
PredInd is not a variable and it is not of the form **Name / Arity**.

current_predicate/2

Description

`current_predicate(Mod,PredInd)`

Succeeds if PredInd is a functor for one of the user-defined procedures in the database of module **Mod**. **PredInd** has to be a variable or a functor expression of form **Name / Arity**, without a module prefix.

This predicate is resatisfiable. On backtracking, it gives all the possible functors. If the database is modified after a **current_predicate/2** call, the modifications will not be effective from the point of view of the **current_predicate/2** call.

Each user-defined procedure is found, whether static or dynamic. A user-defined procedure is also found even when all its clauses have been previously retracted.

Template and modes

`current_predicate(+module_name,?predicate_indicator)`

Examples

`current_predicate(mod1, foo/2).`
Succeeds if there is a user-defined procedure **`foo/2`** in module **`mod1`**.

`current_predicate(not_mod, foo/2).`
If there is not module **not_mod** fails.

Errors

`type_error(atom, Mod)`
Mod is not an atom.
`existence_error(module, Mod)`
Mod is not a module name.
`type_error(predicate_indicator, PredInd)`
PredInd is not a variable and it is not of the form **Name / Arity**.

current_static_predicate/2

current_dynamic_predicate/2

Description

`current_static_predicate(Mod,PredInd)`

`current_dynamic_predicate(Mod,PredInd)`

These predicates are similar to

```
current_predicate(Mod,PredInd),
```

but they succeed only for static or dynamic predicates.

Template and modes

```
current_static_predicate(+mod_name,?predicate_indicator)
current_dynamic_predicate(+mod_name,?predicate_indicator)
```

Examples

`current_static_predicate(mod1, foo/2).`
Succeeds if there is a user-defined static predicate **`foo/2`** in module **`mod1`**.

`current_dynamic_predicate(mod1, dyn_foo/2).`
Succeeds if there is a user-defined dynamic predicate **`dyn_foo/2`** in module **`mod1`**.

`current_static_predicate(mod1, X).`
Succeeds unifying **X** with functor of a user_defined static predicate in module **mod1**. On backtrack all possible functors are found.

Errors

`type_error(atom, Mod)`
Mod is not an atom.
`existence_error(module, Mod)`
Mod is not a module name.
`type_error(predicate_indicator, PredInd)`
PredInd is not a variable and it is not of the form **Name / Arity**.

current_standard_predicate/1

Description

`current_standard_predicate(PredInd)`

Succeeds if **PredInd** is the functor of a built-in (standard) predicate. **PredInd** has to be a variable or a functor expression of form **Name / Arity**, without a module prefix.

This predicate is resatisfiable. On backtracking, it gives all the possible functors.

Template and modes

`current_standard_predicate(?predicate_indicator)`

Examples

`current_standard_predicate(is/2).`
Succeeds
`current_standard_predicate(foo/2).`
Fails.

Errors

`type_error(predicate_indicator, PredInd)`
PredInd is not a variable and it is not of the form **Name / Arity**.

current_module/1

Description

`current_module(Mod)`

Succeeds if **Mod** is the name of a user-defined module in the program. This predicate is resatisfiable, when it is called with a variable argument it returns all the module names.

Template and modes

`current_module(?module_name)`

Examples

`current_module(mod)`
Succeeds if **mod** is the name of a user defined module.

Errors

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

18. Clause creation and destruction

These predicates enable the Prolog database to be altered during execution.

All database modifying built-in predicates have a backtrackable version. These predicates have the same behavior, but if a backtrack reaches them, all the global effect of predicates is undone. So, if a clause was added, it is removed; and if a clause was removed, it is re-added. These predicates have the same names as their non-backtrackable variants, with a suffix **_b**.

The non-backtrackable and backtrackable built-in predicates cannot be mixed when dealing with the same dynamic procedure. That means that the first operation on a dynamic procedure decides whether it will be handled in backtrackable or non-backtrackable way, and this decision cannot be changed later.

It is an interesting question what happens if during the execution of a dynamic predicate the predicate itself is changed and then backtrack occurs. If some clauses had been added or removed, are they found by the backtrack or they are not? The solution of CS-Prolog for this problem is the so-called logical view of dynamic calls. It means that the definition of a dynamic predicate that is called is effectively frozen when the call is made. For a call of a dynamic predicate, it will always contain exactly the same clauses it contained when the call was made.

This logical view of dynamic calls can be explained better if we imagine that a call of a dynamic procedure makes a virtual copy of the predicate, then executes the copy rather than the original procedure. Any changes to the predicate are immediately reflected in the Prolog database, but not in the copy being executed. Therefore, changes to an executed predicate will not be effective on backtracking. A subsequent call, however, makes and executes a virtual copy of the modified Prolog database.

Note that any dynamic clause always belongs to one particular process; no other process has access to it. Different processes can have different clauses with the same functor.

asserta/1

assertz/1

Description

`asserta(Clause)`

`assertz(Clause)`

Constructs a clause from the term **Clause** and adds the new clause to the database to the dynamic predicate whose functor matches the functor of the head of **Clause**. The clause is added before all existing clauses in case of **asserta/1** and after all existing clauses in case of **assertz/1**. **Clause** can contain a module prefix; in this case the clause is added into that module. Otherwise the current module is updated.

Template and modes

`asserta(@clause)` [ISO]
`assertz(@clause)` [ISO]

Examples

```
asserta(mod1:atom).
```
Succeeds.

```
assertz((test(X) :- cond(X), cond2(X))).
```
Succeeds.

Errors

`instantiation_error`
The head of **Clause** or its module prefix is uninstantiated.

`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.

`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.
`type_error(callable, Culprit)`
Either the head of **Clause** is not callable, in which case **Culprit** = **Head**, or **Body** contains a call **Culprit** which is not callable.
`type_error(atom, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is not an atom.
`domain_error(module_name, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is the **nil** atom (not allowed as module name).
`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.
`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, backtrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in backtrackable way.
`representation_error(Flag)`
An implementation-defined limit was exceeded during the compilation of the clause. **Flag** can be one of the following atoms: **max_clause_complexity**, **max_path_in_clause**, **max_call_in_clause**, and **max_call_in_path**.

assertn/2

Description

`assertn(Clause,N)`

Constructs a clause form the term **Clause** and adds it as the **N**`th` clause to the dynamic procedure whose functor matches the functor of the head of **Clause**. **Clause** can contain a module prefix; in this case the clause is added into that module. Otherwise the current module is updated. If **N** is 0 the clause is added as the last one.

Template and modes

`assertn(@clause,+integer)`

Examples

`assertn(mod1:atom,1).`
Succeeds. It is equivalent with **`asserta(mod1:atom)`**.
`assertn((test(X) :- cond(X), cond2(X)), 0).`
Succeeds. It is equivalent with an **`assertz`** call with same first argument.
`assertn( module : (foo :- X), 4).`
Succeeds. The added clause will be the fourth one.

Errors

`instantiation_error`
The head of **Clause** or its module prefix is uninstantiated.
`instantiation_error`
N is a variable
`type_error(integer, N)`
N is not an integer
`domain_error(not_less_than_zero, N)`
N is less than zero.
`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.

`type_error(callable, Culprit)`
Either the head of **Clause** is not callable, in which case **Culprit** = **Head**; or **Body** contains a call **Culprit** that is not callable.
`type_error(atom, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is not an atom.
`domain_error(module_name, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is the **nil** atom (not allowed as module name).
`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.
`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, backtrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in backtrackable way.
`representation_error(Flag)`
An implementation-defined limit was exceeded during the compilation of the clause. **Flag** can be one of the following atoms: **max_clause_complexity**, **max_path_in_clause**, **max_call_in_clause**, and **max_call_in_path**.

retract/1

Description

`retract(Clause)`

Succeeds if the functor of the head in **Clause** corresponds to a dynamic predicate, and there is a clause in the database with term form **H :- B** which unifies with **Clause**. If the clause is a fact then **B** is **true**. This predicate is removed from the database. **Clause** can contain a module name prefix, then the clause is searched there, if it doesn't the current module is searched.

This predicate is resatisfiable. On backtracking removes all matching clauses. If the predicate is modified after a **retract/1** call, the modifications are not effective from the point of view of the **retract/1** call. The predicate behaves as if it were frozen in the moment of the call.

If the functor of the head in **Clause** corresponds to a static or built-in procedure, **retract/1** raises an exception. If the functor does not correspond to any procedure, the call simply fails.

Template and modes

`retract(+clause)` [ISO]

Examples

Suppose the database of the current module contains the clauses

```
country(switzerland).
country(sweden).
country(X) :-
   in_eec(X).
```

Then

`retract(country(switzerland)).`
Succeeds and removes the first clause for **`country/1`**.

`retract(country(uk)).`
Fails.

`retract((country(uk) :- Tail)).`
Succeeds and removes the third clause for **`country/1`**. **`Tail`** is unified with **`in_eec(uk)`**.

`retract((country(X) :- true)).`
Succeeds and removes the first clause for **`country/1`**. **`X`** is unified with **`switzerland`**. On resatisfaction, succeeds and removes the second clause for **`country/1`**. **`X`** is unified with **`sweden`**.

`retract((country(X) :- Y)).`
Succeeds and removes the first clause for **`country/1`**. **`X`** is unified with **`switzerland`** and **`Y`** with **`true`**. On resatisfaction, succeeds and removes the second clause for **`country/1`**. **`X`** is unified with **`sweden`** and **`Y`** with **`true`**. On resatisfaction, succeeds and removes the third clause for **`country/1`**. **`X`** is unified with a variable and **`Y`** with **`in_eec(X)`**.
`retract((country(X) :- in_eec(Y))).`
Succeeds and removes the third clause for **`country/1`**. **`X`** is unified with **`Y`**.

Errors
`instantiation_error`
Clause head or the module prefix extracted from **Clause** is uninstantiated.
`type_error(callable, Head)`
Clause head extracted from **Clause** is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.
`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.
`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, backtrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in backtrackable way.

retractn/2

Description
`retractn(PredInd,N)`

Removes the **N**th clause of the dynamic predicate with predicate indicator **PredInd**. **PredInd** can contain a module prefix in which case the clause is searched in that module; otherwise the current module is updated.

Template and modes
`retractn(+predicate_indicator,+integer)`

Examples
`retractn(mod1:foo/2,3)`
Removes the third clause of predicate **`foo/2`** in module **`mod1`**.

Errors
`instantiation_error`
PredInd is insufficiently instantiated.
`instantiation_error`
N is a variable
`type_error(integer, N)`
N is not an integer
`domain_error(not_less_than_zero, N)`
N is less than zero.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.

`type_error(predicate_indicator, PS)`
PS, the predicate indicator extracted from **PredInd**, is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS, the predicate indicator extracted from **PredInd**, is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS, the predicate indicator extracted from **PredInd**, is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS, the predicate indicator extracted from **PredInd**, is of the form **Name / Arity**, where **Arity** < 0.
`representation_error(max_arity)`
PS, the predicate indicator extracted from **PredInd**, is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(modify, static_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a static procedure.
`permission_error(modify, control_construct, PS)`
PS, the predicate indicator extracted from **PredInd**, is a Prolog control construct.
`permission_error(modify, builtin_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a built-in procedure.
`existence_error(procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is not a functor of a dynamic procedure. **ErrInfo-Argno** is 1.
`existence_error(procedure, N)`
There is no **N**th clause in predicate. **ErrInfo-Argno** is 2.
`permission_error(modify, backtrackable_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, and this procedure is being handled in backtrackable way.

abolish/1

Description

`abolish(PredInd)`

Removes all clauses of the dynamic predicate whose functor is equal to **PredInd**. **PredInd** can contain a module prefix in which case that module is changed; otherwise the current module is updated. The predicate succeeds even if there are no clauses to be removed.

Template and modes

`abolish(+predicate_indicator)` `[ISO]`

Examples

```
abolish(mod:foo/2).
```
Succeeds and removes all clauses of predicate **`foo/2`** in module **`mod`**.

Errors

`instantiation_error`
PredInd is insufficiently instantiated.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0.

`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(modify, static_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a static procedure.
`permission_error(modify, control_construct, PS)`
PS, the predicate indicator extracted from **PredInd**, is a Prolog control construct.
`permission_error(modify, builtin_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a built-in procedure.
`permission_error(modify, backtrackable_procedure, PS)`
PS is the predicate specification and this procedure was handled in backtrackable way.

asserta_b/1

assertz_b/1

Description

`asserta_b(Clause)`

`assertz_b(Clause)`

Constructs a clause form the term **Clause** and adds it to the database to the dynamic predicate whose functor matches the functor of the head of **Clause**. The clause is added before all existing clauses in case of **asserta_b/1** and after all existing clauses in case of **assertz_b/1**. **Clause** can contain a module prefix, in which case the clause is added into that module. Otherwise the current module is updated.

On backtrack, the clause that was added is removed from the data base.

Template and modes

```
asserta_b(@clause)
assertz_b(@clause)
```

Examples

`asserta_b(mod1:atom).`
Succeeds. When backtracking this clause is removed.
`assertz_b((test(X) :- cond(X), cond2(X))).`
Succeeds. When backtracking this clause is removed.

Errors

`instantiation_error`
The head of **Clause** or its module prefix is uninstantiated.
`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.
`type_error(callable, Culprit)`
Either the head of **Clause** is not callable, in which case **Culprit** = **Head**, or **Body** contains a call **Culprit** which is not callable.
`type_error(atom, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is not an atom.
`domain_error(module_name, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is the **nil** atom (not allowed as module name).
`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.

`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, nonbacktrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in non-backtrackable way.
`representation_error(Flag)`
An implementation-defined limit was exceeded during the compilation of the clause. **Flag** can be one of the following atoms: **max_clause_complexity**, **max_path_in_clause**, **max_call_in_clause**, and **max_call_in_path**.

assertn_b/2

Description

`assertn_b(Clause,N)`

Constructs a clause form the term **Clause** and adds it as the **N**`th` clause of the dynamic predicate whose functor matches the functor of the head of **Clause**. **Clause** can contain a module prefix, in which case the clause is added into that module. Otherwise the current module is updated. If **N** is 0 the clause is added as the last one.

On backtrack, the clause that was added is removed from the data base.

Template and modes

`assertn_b(@clause,+integer)`

Examples

`assertn_b(mod1:atom,1).`
Succeeds. It is equivalent with **`asserta(mod1:atom)`**.
`assertn_b((test(X) :- cond(X), cond2(X)), 0).`
Succeeds. It is equivalent with an **`assertz`** call with same first argument.
`assertn_b( module : (foo :- X), 4).`
Succeeds. The added clause will be the fourth one.

Errors

`instantiation_error`
The head of **Clause** or its module prefix is uninstantiated.
`instantiation_error`
N is a variable
`type_error(integer, N)`
N is not an integer
`domain_error(not_less_than_zero, N)`
N is less than zero.
`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.
`type_error(callable, Culprit)`
Either the head of **Clause** is not callable, in which case **Culprit** = **Head**; or **Body** contains a call **Culprit** that is not callable.
`type_error(atom, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is not an atom.
`domain_error(module_name, Culprit)`
Body contains a prefixed call **Culprit** in which the module name prefix is the **nil** atom (not allowed as module name).
`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.

`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, nonbacktrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in non-backtrackable way.
`representation_error(Flag)`
An implementation-defined limit was exceeded during the compilation of the clause. **Flag** can be one of the following atoms: **max_clause_complexity**, **max_path_in_clause**, **max_call_in_clause**, and **max_call_in_path**.

retract_b/1

Description

`retract_b(Clause)`

Is true if the functor of the head in **Clause** corresponds to a dynamic predicate, and there is a clause in the database with term form **H :- B** that unifies with **Clause**. This predicate is removed from the database. If **Clause** contains a module name prefix, then the clause is searched there, if it doesn't the current module is searched.

This predicate is resatisfiable. On backtracking removes all matching clauses. If the predicate is modified after a **retract_b/1** call, the modifications are not effective from the point of view of the **retract_b/1** call. The predicate behaves as if it was frozen in the moment of the call.

On backtrack, the clause that was removed is added back to the data base.

Template and modes

`retract_b(+clause)`

Examples

Suppose the database of the current module contains the clauses

```
country(switzerland).
country(sweden).
country(X) :-
   in_eec(X).
```

Then

`retract_b(country(switzerland)).`
Succeeds and removes the first clause for **`country/1`**. If this call is backtracked the clause reappears in the database.
`retract_b(country(uk)).`
Fails.
`retract_b((country(uk) :- Tail)).`
Succeeds and removes the third clause for **`country/1`**. **`Tail`** is unified with **`in_eec(uk)`**. If this call is backtracked the clause reappears in the database.
`retract_b((country(X) :- true)).`
Succeeds and removes the first clause for **`country/1`**. **`X`** is unified with **`switzerland`**. If this call is backtracked the clause reappears in the database and the **`retract_b/1`** removes the second clause for **`country/1`**. **`X`** is unified with **`sweden`**.

Errors

`instantiation_error`
Clause head or the module prefix extracted from **Clause** is uninstantiated.
`type_error(callable, Head)`
Clause head extracted from **Clause** is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod module prefix extracted from **Clause** is not an atom.
`existence_error(module, Mod)`
Mod module prefix extracted from **Clause** is not a module name.

`permission_error(modify, static_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a static procedure.
`permission_error(modify, control_construct, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is a Prolog control construct.
`permission_error(modify, builtin_procedure, CulpritFunc)`
CulpritFunc, the principal functor of the head of the clause, is the name of a built-in procedure.
`permission_error(modify, nonbacktrackable_procedure, CulpritFunc)`
CulpritFunc is the principal functor of the head and this procedure was handled in non-backtrackable way.

retractn_b/2

Description

`retractn_b(PredInd,N)`

Removes the **N**th clause of the dynamic predicate whose functor is equal to **PredInd**. **PredInd** can contain a module prefix, in which case the clause is searched in that module; otherwise the current module is updated.

On backtrack, the clause that was removed is added back to the data base.

Template and modes

`retractn_b(+predicate_indicator,+integer)`

Examples

`retractn_b(mod1:foo/2,3)`
Removes the third clause of predicate foo/2 in module mod1.

Errors

`instantiation_error`
PredInd is insufficiently instantiated.
`instantiation_error`
N is a variable
`type_error(integer, N)`
N is not an integer
`domain_error(not_less_than_zero, N)`
N is less than zero.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0.
`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(modify, static_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a static procedure.
`permission_error(modify, control_construct, PS)`
PS, the predicate indicator extracted from **PredInd**, is a Prolog control construct.
`permission_error(modify, builtin_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a built-in procedure.

`existence_error(procedure, PS)`
PS is the predicate specification and it is not a functor of a dynamic procedure. **ErrInfo-Argno** is 1.
`existence_error(procedure, N)`
There is no **N**th clause in predicate. **ErrInfo-Argno** is 2.
`permission_error(modify, nonbacktrackable_procedure, PS)`
PS is the predicate specification and this procedure was handled in non-backtrackable way.

abolish_b/1

Description

`abolish_b(PredInd)`

Removes all clauses of the dynamic predicate whose functor is equal to **PredInd**. **PredInd** can contain a module prefix, in which case that module is changed; otherwise the current module is updated.

On backtrack, the clauses that was removed are added back to the data base.

Template and modes

`abolish_b(+predicate_indicator)`

Examples

`abolish_b(mod:foo/2).`
Succeeds and removes all clauses of predicate **`foo/2`** in module **`mod`**.

Errors

`instantiation_error`
PredInd is insufficiently instantiated.
`type_error(atom, Mod)`
Mod module prefix extracted from **PredInd** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0.
`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.
`permission_error(modify, static_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a static procedure.
`permission_error(modify, control_construct, PS)`
PS, the predicate indicator extracted from **PredInd**, is a Prolog control construct.
`permission_error(modify, builtin_procedure, PS)`
PS, the predicate indicator extracted from **PredInd**, is the name of a built-in procedure.
`permission_error(modify, nonbacktrackable_procedure, PS)`
PS is the predicate specification and this procedure was handled in non-backtrackable way.

19. Global value handling

If a term is to be stored outside the evaluation stack for later retrieval only, then using the **assert-retract** predicates is a bit wasteful, because assert generates executable code from the clause, which is not needed in this case. The predicates in this group allow the user to store and retrieve Prolog terms in a more compact non-executable form, associating them with arbitrarily selected **value names**.

Integers and atoms can be used as value names. The expression **table_key** is used in templates and exception terms to indicate that the type of a value-name argument should be integer or atom.

The name **global value** is used to emphasize the fact that the storage structure employed (**value table**, for short) is module-independent, i.e. the same values can be accessed from different modules (but each process has a separate value table).

The value table can contain entries of two distinct kinds: backtackable and non-backtrackable ones. Different predicates are used to create and modify each; some of them operate only on non-backtrackable entries.

The non-backtrackable entries in general maintain a stack-like structure. The values associated with the global name form a stack of individual values (terms). Only the current top of the stack can be accessed (modified or retrieved). Deleting the top of the stack (pop) exposes the previous level of the stack (if there remains any). The terms stored this way are **copies** of the corresponding argument of the predicate that placed them into the value table (cf. **copy_term**).

The backtrackable entries maintain only a single term; there is no stack structure at all. In this case backtracking across a value-modifying predicate call automatically restores the previous state of the entry. The value entry is **the term itself**, not a copy, so the stored value will follow the changes of the term (when any variables get instantiated in it). A backtrackable global value entry can be looked upon as a hidden argument passed down to every called predicate.

Note that values are local to a process. Different processes can have different values stored under the same name.

19.1 Clist values

There are special variants of value terms stored in non-backtrackable entries for accumulating lists incrementally, called **clist** values. They are used internally by the all-solutions predicates, but are made available for application programs, too, as they might be useful in some circumstances. These special list values must be initialized by the specific **push_empty_[k][o]cl_value/2** predicate corresponding to the desired type of the list to be built at the next stack level. The type of the created list is chosen in accordance with the presence or absence of the optional mnemonic letters: **k** for **keyed** and **o** for **ordered**. There are also special **add_to_[k]cl_value/[2;3]** predicates for incrementally adding new items to the list.

Keyed clist serve for partitioning a collection of terms into sublists according to some partition keys explicitly specified in the **add_to_kcl_value/3** predicate (not to be confused with the value name, which is also called **table_key**). Non-empty values of this kind form a two-level list structure. The elements of the upper level list are sublists, where the first (head) item is the specific partition key, and the rest consists of the items added under this particular partition key.

Unkeyed clist values consist of a simple list of the individually added terms.

In an **ordered unkeyed** clist, the accumulated list is an ordered set of the added terms (without duplicates, ordered according to the term-precedence relation, see section 14.1). In an **unordered unkeyed** clist duplicates are not eliminated, and the items are in order of arrival.

For **keyed** clists ordering applies to the partition sublists excepting the partition key placed at the head of the sublist. The partitions themselves are in the order of the first occurrence of the individual partition keys. (As a matter of fact the key does not belong to the sublist, a partition is only for simplicity's sake retrieved as **`[<key>|<itemlist>]`** instead of the more natural **`<key> - <itemlist>`** structure representation.)

When non-ground terms occur as partition key and/or list item, the following are to be taken into account.

Non-backtrackable value items are stored as copies of the passed arguments, where each variable occurring in the term is replaced by a new unnamed variable. In the case of keyed clists, first a temporary term is built that contains both the specified partition key and the item to be stored as its subterms; and a copy of this temporary

term is manipulated further. In this way, those variables that occur in both the key and the item will remain common.

There is a peculiarity in the handling of keyed ordered clists, due to the needs of the **setof/3** built-in predicate. If some partition keys specified in different activations of the **add_to_kcl_value/3** predicate are variants of each other (i.e. can be brought to identical form by systematically renaming their variables) then they are treated as identical.

19.2 Overview of predicate use

Backtrackable values are created and modified by **set_value_b/2**. The generic **get_value/2** can be used for retrieval. Backtrackable values are destroyed implicitly when backtracking occurs across the **set_value_b/2** call that created the value entry.

Non-backtrackable value entry stack levels are managed by predicates from the **push_value** and **pop_value** family. For the sake of similarity with the backtrackable case, the first level can also be created implicitly for an ordinary (non-clist) entry. The value table entry is created when the first **push_value** type call is issued against a currently non-associated table key. The entry is destroyed when its last stack level is removed in response to a **pop_value/1,2** call.

Ordinary non-backtrackable value entries are created by the first **push_value/2** call issued against a currently non-existing value table entry, either explicitly or implicitly from the first **set_value/2** call. Further stack levels are created by subsequent **push_value/2** calls. The value term stored at the current stack level can be changed by **set_value/2**. The term stored at the current level can be retrieved using either **get_value/2,** which leaves the entry unchanged, or **pop_value/2**, which unifies its second argument with the stored term and then removes the current level (top) from the value stack. **pop_value/1** is used to remove the current level without retrieving the term stored there.

Clist values can be stored at any stack level of a non-backtrackable value table entry. The level must be initialized by the appropriate member of the **push_empty_[k][o]cl_value/1** predicate group, which sets the value to the empty list (of the proper kind). The clist value at the current level can be modified (new item added) by the **add_to_cl_value/2** predicate for a non-keyed clist, or the **add_to_kcl_value/2** predicate for a keyed clist. **get_value/2** and **pop_value/1,2** can be used for clist values just as for ordinary ones.

The set of **incr_value/1,2,3** and **incr_value_b/1,2,3** convenience predicates can be used to increase or decrease numeric values stored at the current top stack level of non-backtrackable entries and in backtrackable entries, respectively, in one call.

set_value/2

Description

```
set_value(Name, Value)
```

Stores a copy of the term **Value** under the name **Name** for the current level of a non-backtrackable value. The stored copy can later be retrieved using **get_value/2** or **pop_value/2**. The term **Value** is first copied, so if it contains variables, then in the stored term the variables are substituted by new ones. If the entry for **Name** does not exist, creates it (performs an implicit **push_value/2**), otherwise the previous (ordinary) value at the current level is replaced. In the latter case, if the current stack level contains a clist value, an error occurs.

Template and modes

```
set_value(+table_key,@term).
```

Examples

```
set_value(counter, 0).
```

Associates the value 0 with the name **`counter`**.

```
set_value(data, f(X,Y)).
```

Sets the value **`f(_,_)`** to the name **`data`**. If later on this value is retrieved, the two arguments of the term **`f/2`** will be new variables, different from the original **`X`** and **`Y`**.

Errors

`instantiation_error`
Name is uninstantiated.
`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.
`permission_error(modify, value, Name)`
Name was previously associated with a value through **set_value_b/2** or the current stack level was initialized for a clist term.

push_value/2

Description

`push_value(Name, Value)`

Stores a copy of the term **Value** at the next level of the stack associated with the non-backtrackable value entry named **Name**, and makes that level the current top of the stack. The stored term can be retrieved later using **get_value/2** or **pop_value/2**. The term **Value** is copied first, so any variables occurring in it are replaced by new ones. This predicate creates a new level on the stack of values. If **Name** was assigned previously a value, the previous value is not lost. A subsequent **pop_value/1,2** will restore that value.

If there is no value table entry associated with **Name** at the time of the call, then a new (non-backtrackable) entry is created for **Name** and the term is stored at the first stack level.

Template and modes

`push_value(+table_key,@term).`

Examples

`push_value(counter, 11).`
Sets the value 11 to the name **`counter`**. After a later **`pop_value(counter)`** call the previous value of **`counter`** is restored.

Errors

`instantiation_error`
Name is uninstantiated.
`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.
`permission_error(modify, value, Name)`
Name was previously associated with a value through **set_value_b/2**.

pop_value/1

pop_value/2

Description

`pop_value(Name)`

Is equivalent with

```
pop_value(Name,_)
```

`pop_value(Name,Term)`

Unifies Term with the term stored at the current level of the value stack associated with **Name**, then removes the current level from the stack. If this level was the last one then the value table entry is destroyed and **Name** becomes dissociated. Otherwise, the previous stack level becomes the current top (the value stored before the last **push** is restored).

Template and modes

```
pop_value(+table_key)
pop_value(+table_key, ?term).
```

Examples

```
pop_value(counter).
```
Restores the value of **counter** to the state that it had before the last **push_value/2** call.

Errors

`instantiation_error`
Name is uninstantiated.
`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.
`permission_error(access, value, Name)`
There is no value associated with table key **Name**.
`permission_error(modify, value, Name)`
Name currently is associated with a backtrackable value table entry.

delete_value/1

Description

```
delete_value(Name)
```

Deletes the non-backtrackable value table entry associated with **Name**. All stored terms at any level of the stack are lost. **Name** becomes a non value name (dissociated).

Template and modes

```
delete_value(+table_key).
```

Examples

```
delete_value(counter).
```
Deletes all values of **counter**, all subsequent retrieving of value associated to this name will cause an exception.

Errors

`instantiation_error`
Name is uninstantiated.
`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.
`permission_error(modify, value, Name)`
Name is currently associated with a backtrackable value table entry (via **set_value_b/2**).

get_value/2

Description

```
get_value(Name, Term)
```

Retrieves the term stored under the name **Name** and unifies it with **Term**. This predicate can be used for any kind of value table entries. For non-backtrackable entries, the term stored on the current top level is retrieved. The state of the entry remains unchanged.

Template and modes

```
get_value(+table_key, ?term).
```

Examples

```
get_value(counter,X).
```

Unifies **X** with the value last associated with the value name **counter**.

Errors

```
instantiation_error
```

Name is uninstantiated.

```
type_error(table_key, Name)
```

Name is not a variable and not an atom or integer.

```
permission_error(access, value, Name)
```

There is no value assigned to table key **Name**.

test_value/1

test_value/2

Description

```
test_value(Name)
```

is equivalent with

```
test_value(Name, _)
```

```
test_value(Name, Term)
```

If there is no current value associated with **Name** then the predicate fails. Otherwise it is the same as **get_value/2**; i.e. retrieves the term stored under the name **Name** and unifies it with **Term**.

Template and modes

```
test_value(+table_key).
test_value(+table_key, ?term).
```

Examples

```
test_value(optional).
```

If the name **optional** has no value associated with it (not set previously or deleted) then fails, otherwise succeeds.

```
test_value(counter,X).
```

Unifies **X** with the value last associated with the value name **counter**, or fails if no value is currently associated with **counter**.

Errors

```
instantiation_error
```

Name is uninstantiated.

```
type_error(table_key, Name)
```

Name is not a variable and not an atom or integer.

set_value_b/2

Description

```
set_value_b(Name, Value)
```

Stores the term **Value** in a backtrackable value table entry under the name **Name**. The stored term can be later retrieved using **get_value/2**. The **Value** term is not copied, so if it contains variables, then in the stored term these variables are the same ones that are in **Value**. If such a variable becomes later instantiated, the stored term will follow the changes.

If **Name** was assigned previously a value, that previous value will be restored when backtracking across this call occurs.

Template and modes

```
set_value_b(+table_key, @term).
```

Examples

```
set_value_b(counter, 0).
```
Sets the value 0 to the name **counter**. When backtracking on this call, the previous value will be restored, or if there was no value associated to **counter** previously, then this name will no longer be a value name.

```
set_value_b(data, f(X,Y)).
```
Sets the value **f(X,Y)** to the name **data**. If later on this value is retrieved, the two arguments of the term **f/2** will be the same variables as **X** and **Y**. When backtracking on this call, the previous value, if any, will be restored.

Errors

```
instantiation_error
```
Name is uninstantiated.

```
type_error(table_key, Name)
```
Name is not a variable and not an atom or integer.

```
permission_error(modify, value, Name)
```
Name is currently associated with a non-backtrackable value table entry.

incr_value/1

incr_value /2

incr_value /3

Description

```
incr_value(Name)
```

Is equivalent with

```
incr_value(Name, 1)
```

```
incr_value(Name, Delta)
```

Is equivalent with

```
incr_value(Name, Delta, _)
```

```
incr_value(Name, Delta, PrevValue)
```

Increments the number stored under the name **Name** with **Delta**. **PrevValue** is unified with the previous value. This unification takes place before storing the incremented value, so if it fails, the new (incremented) value is not stored at all.

The effect of **incr_value(Name, Delta, Prev_value)** can be described by the following procedure definition:

```
incr_value(Name, Delta, Prev_value) :-
            get_value(Name, Prev_value),
                  V is Prev_value + Delta,
                  set_value(Name, V).
```

Template and modes

```
incr_value(+table_key).
incr_value(+table_key, +number, @term).
incr_value(+table_key, +number, @term).
```

Examples

```
incr_value(counter, 12).
```

Increments the value associated with the name **counter** by 12.

Errors

```
instantiation_error
```

Name is uninstantiated.

```
type_error(table_key, Name)
```

Name is not a variable and not an atom or integer.

```
permission_error(access, value, Name)
```

There is no value assigned to table key **Name**.

```
permission_error(modify, value, Name)
```

The value assigned to table key **Name** is not a number. **ErrInfo-Other** will contain this non-numeric value.

```
instantiation_error
```

Delta is uninstantiated.

```
type_error(number, Delta)
```

Delta is not a variable and not an integer or float.

```
permission_error(modify, value, Name)
```

Name is currently associated with a backtrackable value table entry.

```
evaluation_error(Flag)
```

The addition caused an overflow error. **Flag** is one of the following flags: **int_overflow**, **float_overflow**.

incr_value_b/1

incr_value_b/2

incr_value_b/3

Description

```
incr_value_b(Name)
```

Is equivalent with

```
incr_value_b(Name, 1)
```

```
incr_value_b(Name, Delta)
```

Is equivalent with

```
incr_value_b(Name, Delta, _)
```

```
incr_value_b(Name, Delta, PrevValue)
```

Increments the number stored under the name **Name** with **Delta**. **PrevValue** is unified with the previous value. This unification takes place before storing the incremented value, so if it fails, the new (incremented) value is not stored at all. The value modification is backtrackable, so the previous value is restored if backtracking across this call occurs.

The effect of **incr_value(Name, Delta, Prev_value)** can be described by the following procedure definition:

```
incr_value_b(Name, Delta, Prev_value) :-
            get_value(Name, Prev_value),
                  V is Prev_value + Delta,
                  set_value_b(Name, V).
```

Template and modes

```
incr_value_b(+table_key).
incr_value_b(+table_key, +number, @term).
incr_value_b(+table_key, +number, @term).
```

Examples

```
incr_value_b(counter, 12).
```

Increments the value associated with the name **counter** by 12.

Errors

```
instantiation_error
```

Name is uninstantiated.

```
type_error(table_key, Name)
```

Name is not a variable and not an atom or integer.

```
permission_error(access, value, Name)
```

There is no value assigned to table key **Name**.

```
permission_error(modify, value, Name)
```

The value assigned to table key **Name** is not a number. **ErrInfo-Other** will contain this non-numeric value.

```
instantiation_error
```

Delta is uninstantiated.

```
type_error(number, Delta)
```

Delta is not a variable and not an integer or float.

```
permission_error(modify, value, Name)
```

Name is currently associated with a non-backtrackable value table entry.

```
evaluation_error(Flag)
```

The addition caused an overflow error. **Flag** is one of the following flags: **int_overflow**, **float_overflow**.

push_empty_cl_value/1

push_empty_ocl_value/1

push_empty_kcl_value/1

push_empty_kocl_value/1

Description

All predicates in this group initialize the next stack level of the non-backtrackable value table entry associated with **Name** (the only argument of the call) to an empty clist value, the exact kind of which depends on the predicate called. If no entry is currently associated with **Name** then the call creates one for it and initializes the first stack level. The clist value stored at the top level can be changed by adding list items to it in subsequent **add_to_[k]cl_value/2** calls. The stored clist term can be retrieved at any time using **get_value/2** (as long as it is at the current top). The retrieved value is a normal term equivalent with the stored clist structure. The level can be removed by **pop_value/1,2**, where **pop_value/2** also retrieves the value stored at the level just removed. All of these can be looked upon as special variants of the

```
push_value(Name, [])
```

call, the difference being only in the subsequent value-modifying calls allowed.

The variations in the effect of the individual predicates are as follows:

```
push_empty_cl_value(Name)
```

Initializes the stack level to an empty **unkeyed unordered** clist value. List items can be added to the current clist value by **add_to_cl_value/2** calls.

```
push_empty_ocl_value(Name)
```

Initializes the stack level to an empty **unkeyed ordered** clist value. List items can be added to the current clist value by **add_to_cl_value/2** calls.

```
push_empty_kcl_value(Name)
```

Initializes the stack level to an empty **keyed unordered** clist value (for accumulating lists of terms under specified partition keys). Items can be added to the current clist value by **add_to_kcl_value/3** calls.

```
push_empty_kocl_value(Name)
```

Initializes the stack level to an empty **keyed ordered** clist value (for accumulating ordered sets of terms under specified partition keys). Items can be added to the current clist value by **add_to_kcl_value/3** calls.

Template and modes

```
push_empty_cl_value(+table_key).
push_empty_ocl_value(+table_key).
push_empty_kcl_value(+table_key).
push_empty_kocl_value(+table_key).
```

Examples

```
push_empty_cl_value(bag).
```
Associates an empty unordered unkeyed clist with the name **bag**.

Errors

`instantiation_error`
Name is uninstantiated.

`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.

`permission_error(access, value, Name)`
There is no value assigned to table key **Name**.

`permission_error(modify, value, Name)`
Name is currently associated with a non-backtrackable value table entry.

add_to_cl_value/2

Description

```
add_to_cl_value(Name,Term)
```

This predicate can be used to change the unkeyed (ordered or unordered) clist term stored at the current top level of the stack of the non-backtrackable value table entry associated with the table-key **Name**.

In the case when an unordered clist is being modified, a copy of **Term** is simply appended to the list stored as clist.

When an ordered clist is to be changed, a new term T1 is built as a copy of **Term**, and if the list already contains an item which is equal with T1 (under term comparison) then the clist remains unchanged (duplicate elimination). Otherwise, T1 is inserted at the position determined by the term precedence relation so that the list is in increasing order. Note that if T1 contains any variables then no item already on the list can be equal with it because of the variable renaming during copy.

Template and modes

```
add_to_cl_value(+table_key, ?term).
```

Examples

```
add_to_cl_value(bag,foo(a,b)).
```
Inserts a new element to the unkeyed clist associated with the name **bag**.

Errors

`instantiation_error`
Name is uninstantiated.

`type_error(table_key, Name)`
Name is not a variable and not an atom or integer.

`permission_error(access, value, Name)`
There is no value assigned to table key **Name**.

`permission_error(modify, value, Name)`
Name was previously associated with a value using some value-handling predicate other then the appropriate **push_empty_cl_value/1** or **push_empty_ocl_value/1** predicate.

add_to_kcl_value/3

Description

```
add_to_kcl_value(Name, Term, Partition_key)
```

This predicate can be used to change the keyed (ordered or unordered) clist term stored at the current top level of the stack of the non-backtrackable value table entry associated with the table-key **Name**. Keyed clists are used to collect separate unordered lists or ordered sets of items grouped according to explicitly specified partition keys.

In both the ordered and unordered case at first a temporary term is built containing **Term** and **Partition_key** as its subterms, and a copy of this is prepared for further manipulation. In this way, common variables occurring in the original arguments remain common after copying.

Next the **Partition_key** portion of the copy is separated as **K1**, and the currently stored clist value is looked up to see whether or not it already contains a sublist with a partition key which is equivalent with, or is a variant of, **K1**. If such a sublist is found, then the rest of the sublist is chosen as the target for insertion, otherwise a new sublist is appended to the clist with **K1** as its head item and the empty list as its tail, and this empty tail is selected as the target.

The final action is the incorporation of **T1**, the **Term** portion of the prepared copy, into the target list. This is done exactly the same way as in the case of **add_to_cl_value/3**, only the target lists are different.

Template and modes

```
add_to_kcl_value(+table_key, ?term, ?term).
```

Examples

```
add_to_cl_value(set,foo(a,b),bar).
```
Inserts a new element with partition key **bar** to the keyed list associated with the name **bag**.

Errors

```
instantiation_error
```
Name is uninstantiated.
```
type_error(table_key, Name)
```
Name is not a variable and not an atom or integer.
```
permission_error(access, value, Name)
```
There is no value assigned to table key **Name**.
```
permission_error(modify, value, Name)
```
Name was previously associated with a value using some value-handling predicate other then the appropriate **push_empty_kcl_value/1** or **push_empty_kocl_value/1** predicate.

20. Binding flexible predicates

These predicate change the bindings that link a flexible imported predicate to another predicate (see section 2.4). Bindings are process-specific. In this respect, flexible predicates are like dynamic predicates (but there is no static initialization for them). A flexible predicate can be bound to another flexible predicate, but such bindings must not form a closed cycle.

When a flexible predicate is unbound, a call for it raises an exception regardless of the current state of the **unknown** Prolog flag.

When a flexible predicate is bound to another predicate (compatible with it, i.e., with the same arity), then a call is equivalent with the call of the predicate which is the target of the binding.

bind/2

Description

`bind(PredInd,Imp)`

Links the flexible predicate with predicate indicator **PredInd** to the predicate specified by **Imp**. After establishing the link, a call of **PredInd** will be equivalent with a call of the link target.

Both **PredInd** and **Imp** can have module prefix for direct import. **Imp** can be given as an atom (after removing the optional module prefix), in which case it is conceptually augmented with the arity of **PredInd** to yield a complete predicate indicator. Otherwise, **Imp** must be a predicate indicator compatible with **PredInd**.

If **PredInd** is a partially flexible predicate, that is, the module of the target predicate is explicitly defined in the corresponding **import** directive, then **Imp** must denote a predicate in that module which is also the default for it without explicit prefixing.

Both arguments must specify items visible from the place of the **bind/2** call, with one extension: if **Imp** is partially flexible, then a local predicate from the restriction module also can be bound to, even if it is not normally visible at the place of the call. (This is for maintaining compatibility with earlier releases).

PredInd can be an unbound flexible predicate or a bound one (with a previous **bind/2**). In the latter case, the previous link is removed.

Imp can also be a flexible predicate, but the requested binding must not create a closed loop of circular links.

Template and modes

```
bind(+predicate_indicator, +atom_or_predicate_indicator).
```

Examples

`bind(foo/2,oof).`
The predicate **`foo/2`** in the current module is bound to the predicate **`oof/2`** in the same module.

`bind(mod:store/2,set_value/2).`
The predicate **`store/2`** in the module **`mod`** is bound to the standard predicate **`set_value/2`**.

Errors

`instantiation_error`
PredInd or **Imp** is insufficiently instantiated.

`type_error(atom, Mod)`
Mod predicate specification, extracted from **PredInd** is not an atom. **ErrInfo-Argno** is 1.

`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name. **ErrInfo-Argno** is 1.

`type_error(atom, Mod)`
Mod predicate specification, extracted from **Imp** is not an atom. **ErrInfo-Argno** is 2.

`existence_error(module, Mod)`
Mod extracted from **Imp** is not a module name. **ErrInfo-Argno** is 2.

`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**. **ErrInfo-Argno** is 1.
`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom. **ErrInfo-Argno** is 1.
`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer. **ErrInfo-Argno** is 1.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0. **ErrInfo-Argno** is 1.
`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255. **ErrInfo-Argno** is 1.
`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **Imp** is not of the form **Name / Arity**. **ErrInfo-Argno** is 2.
`type_error(atom, Name)`
PS predicate specification, extracted from **Imp** is of the form **Name / Arity**, where **Name** is not an atom. **ErrInfo-Argno** is 2.
`type_error(integer, Arity)`
PS predicate specification, extracted from **Imp** is of the form **Name / Arity**, where **Arity** is not an integer. **ErrInfo-Argno** is 2.
`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **Imp** is of the form **Name / Arity**, where **Arity** < 0. **ErrInfo-Argno** is 2.
`representation_error(max_arity)`
PS predicate specification, extracted from **Imp** is of the form **Name / Arity**, where **Arity** > 255. **ErrInfo-Argno** is 2.
`existence_error(procedure, PS)`
PS is the predicate specification extracted from **PredInd** and this functor does not belong to a procedure visible at the place of the call. **ErrInfo-Argno** is 1.
`permission_error(bind, non_flex, PS)`
PS is the predicate specification extracted from **PredInd** and it is the functor of a non-flexible procedure.
`permission_error(bind, part_flex, PS)`
PS denotes a partially flexible predicate with fixed module. The module prefix, extracted from **Imp** is not equal to this module.
`permission_error(bind, wrong_procedure, Arity)`
Arity of **PredInd** is not equal to the arity **Arity** extracted from **Imp**.
`existence_error(procedure, PS)`
PS is the predicate specification extracted from **Imp** and this functor does not belong to a procedure. **ErrInfo-Argno** is 2.
`permission_error(bind, local_procedure, PS)`
PS is the predicate specification extracted from **Imp** and it is the functor of a local (non-exported) procedure not visible from the place of the call.
`permission_error(bind, would_create_loop, PS)`
Imp is also a flexible predicate, currently bound (directly or indirectly) to **PredInd**. Creating the requested binding would close a binding cycle.

unbind/1

Description

`unbind(PredInd)`

The current linkage of the flexible predicate **PredInd** to another predicate is removed. After unbinding, the calls of **PredInd** predicate will cause an **existence_error** exception.

Template and modes

`unbind(+predicate_indicator)`

Examples

```
unbind(foo/2).
```

The flexible predicate **foo/2** in the current module is unbound, from this moment it is an undefined predicate.

Errors

`instantiation_error`
PI is insufficiently instantiated.

`type_error(atom, Mod)`
Mod predicate specification, extracted from **PredInd** is not an atom.

`existence_error(module, Mod)`
Mod extracted from **PredInd** is not a module name.

`type_error(predicate_indicator, PS)`
PS predicate specification, extracted from **PredInd** is not of the form **Name / Arity**.

`type_error(atom, Name)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Name** is not an atom.

`type_error(integer, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** is not an integer.

`domain_error(not_less_than_zero, Arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** < 0.

`representation_error(max_arity)`
PS predicate specification, extracted from **PredInd** is of the form **Name / Arity**, where **Arity** > 255.

`existence_error(procedure, PS)`
PS is the predicate specification extracted from **PredInd** and this functor does not belong to a procedure.

`permission_error(unbind, non_flex, PS)`
PS is the predicate specification extracted from **PredInd** and it is the functor of a non-flexible procedure.

21. File selection and control

This chapter describes the input/output predicates. The basic notions used here are explained in chapter 3.

current_input/1

current_output/1

Description

`current_input(Stream)`

Unifies **Stream** with the current input stream.

`current_output(Stream)`

Unifies **Stream** with the current output stream.

Template and modes

```
current_input(?stream)                                    [ISO]
current_output(?stream)                                   [ISO]
```

Examples

`current_input(X).`
X is unified with the stream identifier of the current input stream.

Errors

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

set_input/1

Description

`set_input(S_or_a)`

Sets the stream associated with stream or alias **S_or_a** to be the current input stream.

set_input/1 cannot fail. Either it succeeds or it raises an exception, in which case the current input stream remains unchanged.

Template and modes

```
set_input(@stream_or_alias)                               [ISO]
```

Examples

`set_input(user_input).`
The current input stream will be the terminal.

Errors

`instantiation_error`
S_or_a is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.

`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).

set_output/1

Description

`set_output(S_or_a)`

Sets the stream associated with stream or alias **S_or_a** to be the current output stream.

set_output/1 cannot fail. Either it succeeds or it raises an exception, in which case the current output stream remains unchanged.

Template and modes

`set_output(@stream_or_alias)` `[ISO]`

Examples

`set_output(user_output).`
The current output stream will be the terminal.

Errors

`instantiation_error`
S_or_a is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).

open/3

Description

`open(Source_sink, Mode, Stream)`

Opens the source/sink **Source_sink** for input or output as indicated by I/O mode **Mode**. Unifies **Stream** with a term representing the stream. **Stream** must be an uninstantiated variable.

This predicate is equivalent with a call of **open/4** with same first three arguments, and an empty list supplied as fourth argument.

Template and modes

`open(@source_sink, @io_mode, -stream)` `[ISO]`

Examples

`open(csp_file, read, S).`
Opens the text file **`'csp_file'`** for reading, and unifies S with a value which will uniquely identify the stream until it is closed.

Errors

`instantiation_error`
Source_sink is a variable.
`instantiation_error`
Mode is a variable.

`domain_error(source_sink, Source_sink)`
Source_sink is not a variable and does not represent a valid source/sink.
`type_error(atom, Mode)`
Mode is not a variable and not an atom.
`type_error(variable, Stream)`
Stream is not an unbound variable.
`domain_error(io_mode, Mode)`
The atom **Mode** is not a valid I/O mode.
`existence_error(source_sink, Source_sink)`
The file or other source/sink specified by **Source_sink** does not exist. **ErrInfo-Other** will be a number, returned as error code by the operating system.
`permission_error(open, source_sink, Source_sink)`
The file or other source/sink specified by **Source_sink** cannot be opened because the operating system does not allow it. **ErrInfo-Other** will be a number, returned as error code by the operating system.

open/4

Description

`open(Source_sink, Mode, Stream, Options)`

Opens the source/sink **Source**_sink for input or output, as indicated by I/O mode **Mode** (see section 3.2) and the list of I/O options **Options** (see section 3.8). Unifies **Stream** with a term representing the stream. **Stream** must be an uninstantiated variable.

Template and modes

`open(@source_sink, @io_mode, -stream, @io_options)  [ISO]`

Examples

`open(csp_file, read, S, [alias(fi)]).`
Opens the text file **`'csp_file'`** for reading, and unifies **`S`** with a value which will uniquely identify the stream until it is closed. The atom **`fi`** is associated as alias to this stream.
`open(memo, write, S, [alias(mem),stream_type(memory)]).`
Opens a memory stream, associates the atom **`mem`** as alias to it, and unifies **`S`** with its stream identifier.

Errors

`instantiation_error`
Source_sink is a variable.
`instantiation_error`
Mode is a variable.
`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.
`domain_error(source_sink, Source_sink)`
Source_sink is not a variable and does not represent a valid source/sink.
`type_error(atom, Mode)`
Mode is not a variable and not an atom.
`type_error(variable, Stream)`
Stream is not an unbound variable.
`type_error(list, Options)`
Options is neither a list nor a partial list.
`domain_error(io_mode, Mode)`
The atom **Mode** is not a valid I/O mode.
`domain_error(stream_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid stream option.

`domain_error(stream_option, Option)`
Option, an element of the **Options** list, is a valid stream option but contradicts to a previous argument. **ErrInfo-Other** will be the atom: **contradiction**.
`existence_error(source_sink, Source_sink)`
The file or other source/sink specified by **Source_sink** does not exist. **ErrInfo-Other** will be a number, returned as error code by the operating system.
`permission_error(open, source_sink, Source_sink)`
The file or other source/sink specified by **Source_sink** cannot be opened because the operating system does not allow it. **ErrInfo-Other** will be a number, returned as error code by the operating system.

reopen/3

Description

`reopen(S_or_a, Mode, Options)`

First closes the stream associated with stream or alias **S_or_a**. Then opens the same stream for input or output as indicated by I/O mode **Mode** and the list of I/O options **Options**. The original alias (if any) cannot be changed and it remains valid even if the **Options** list does not contain such an option.

Template and modes

`reopen(@stream_or_alias, @io_mode, @io_options)`

Examples

`reopen(memo, write, []).`
If **memo** is an alias of a stream, closes that stream. Then opens the source/sink associated with this stream for writing.

Errors

`instantiation_error`
S_or_a is a variable.
`instantiation_error`
Mode is a variable.
`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`type_error(atom, Mode)`
Mode is not a variable and not an atom.
`type_error(list, Options)`
Options is neither a list nor a partial list.
`domain_error(io_mode, Mode)`
The atom **Mode** is not a valid I/O mode.
`domain_error(stream_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid stream option.
`domain_error(stream_option, Option)`
Option, an element of the **Options** list, is a valid stream option but contradicts to a previous argument. If there was an alias associated with the stream previously, a new alias as a stream option leads to this exception. **ErrInfo-Other** will be the atom: **contradiction**.
`permission_error(open, source_sink, Source_sink)`
The file or other source/sink specified by **S_or_a** cannot be opened because the operating system does not allow it. **ErrInfo-Other** will be a number, returned as error code by the operating system.

close/1

close/2

Description

`close(S_or_a)`

Is equivalent with

```
close(S_or_a, []).
```

`close(S_or_a, Options)`

Closes the stream associated with stream or alias **S_or_a** if it is open. The default behavior of this predicate may be modified by specifying a non-empty list of close options in the **Options** parameter.

Valid close options are:

force(true)
force(false)
force

The **force** option is equivalent with **force(true)** . The default is **force(false**). If **force(false)** option is given (the empty option list means this case), and if an error occurs, the stream is not closed and an exception is raised. If **force(true)** option is given then any resource error or system error occurring in the closing of the stream is ignored. Instead of raising an exception in these cases, close forces the stream to be closed and then succeeds.

If the closed stream was the current input or output stream the standard stream becomes the current one.

close/2 cannot fail.

Template and modes

```
close(@stream_or_alias)                                    [ISO]
close(@stream_or_alias, @close_options)                    [ISO]
```

Examples

`close(fi).`
Closes the file associated with alias **`fi`**.

`close(fi,[force(true)]).`
Closes the file associated with alias **`fi`**. It will succeed even if the file cannot be closed because of lack of sufficient space on the disk.

Errors

`instantiation_error`
S_or_a is a variable.

`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.

`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.

`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.

`type_error(list, Options)`
Options is neither a list nor a partial list.

`domain_error(stream_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid close option.

`resource_error(disk_space)`
The close operation cannot be completed because of lack of sufficient space on an external storage device such as a disk to hold the entire sink.

`system_error`
The close operation cannot be completed for some reason other than the above. **ErrInfo-Other** will be a number, returned as error code by the operating system.

flush_output/0

flush_output/1

Description

`flush_output`

Equivalent with

```
current_output(Stream), flush_output(Stream).
```

`flush_output(S_or_a)`

Any output that is currently buffered by the system for the stream associated with stream or alias **S_or_a** is sent to that stream.

Template and modes

```
flush_output                                                    [ISO]
flush_output(@stream_or_alias)                                  [ISO]
```

Examples

`flush_output(user_output).`
All buffered output for the **`user_output`** stream is written out.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).

flush_input/0

flush_input/1

Description

`flush_input`

Equivalent with

```
current_input(Stream), flush_input(Stream)
```

`flush_input(S_or_a)`

Any input that is currently buffered by the system for the stream associated with stream or alias **S_or_a** is deleted, and the buffer is emptied.

In case of files this operation cannot be very useful, since the user has no access to the input buffer. However, in case of the **user_input** - the terminal - it can be helpful to remove the characters typed in the last line.

Template and modes

```
flush_input
flush_input(@stream_or_alias)
```

Examples

`flush_input(user_input).`
All buffered input for the **`user_input`** stream is cleared.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).

stream_property/2

Description

`stream_property(Stream, Property)`

Succeeds if the **Stream** has stream property **Property**. This predicate is resatisfiable. On backtracking, succeeds with all unifiable **Stream**, **Property** pairs.

If the stream properties are modified after a **stream_property/2** call, the modifications are not effective from the point of view of the **stream_property/2** call. The predicate behaves as if the property set was frozen in the moment of the call.

Template and modes

`stream_property(?stream, ?stream_property)` [ISO]

Examples

`stream_property(S, file_name(F)).`
If **S** is instantiated, return the name of the file to which it is connected. Otherwise, backtrack through all streams connected to files and return the file names in **F** and the stream identifiers in **S**.
`stream_property(S, output).`
If **S** is instantiated, check whether output is permitted on this stream. Otherwise, backtrack through all streams currently open for output and unify **S** with their stream identifier.

Errors

`domain_error(stream, Stream)`
Stream is not a variable and it is not a valid stream-term.
`domain_error(stream_property, Property)`
Property is not a variable and it is not a valid stream-property.

at_end_of_stream/0

at_end_of_stream/1

Description

`at_end_of_stream`

Is equivalent with

```
current_input(Stream), at_end_of_stream(Stream)
```

`at_end_of_stream(S_or_a)`

Is true if the stream associated with stream or alias **S_or_a** is positioned at its end or is **past_end_of_stream**.

This predicate succeeds when all the characters in the current input stream or in stream **S_or_a** have been read by input routines (such as **get_code**, **get_char** or **read**), or when stream_position has been used to move

directly to the end of the stream. The predicate **at_end_of_stream/[0,1]** still succeeds when called in the **past_end_of_stream** state.

Template and modes

```
at_end_of_stream                                    [ISO]
at_end_of_stream(@stream_or_alias)                  [ISO]
```

Examples

```
at_end_of_stream(fi).
```

Succeeds if the position for the next read operation in the stream with alias **fi** is at the end of stream.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.

set_stream_position/2

Description

```
set_stream_position(S_or_a,Position)
```

Changes the position of the stream associated with stream or alias **S_or_a** to **Position**. **Position** have previously been returned as a **position/1** stream property of the stream by the predicate **stream_property/2**.

Template and modes

```
set_stream_position(@stream_or_alias, @stream_position)
                                                    [ISO]
```

Examples

```
stream_property(S,position(P)), get_char(CH),
                    set_stream_position(S,P).
```

Reads a character from the stream identified by stream identifier **S**, and then repositions the input stream to its original position.

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Position is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`domain_error(stream_position, Position)`
Position is neither a variable and nor a valid representation of a stream position.
`permission_error(reposition, stream, S_or_a)`
Re-positioning is not allowed on this stream.

get_char/1

get_char/2

Description

`get_char(Char)`

Is equivalent with

```
current_input(Stream), get_char(Stream)
```

`get_char(S_or_a, Char)`

Reads a character from the text stream associated with **S_or_a** and then succeeds if **Char** unifies with the atom consisting of that character. Note that the next character is always removed from the source stream, even if the predicate fails.

Template and modes

```
get_char(?in_character)                                    [ISO]
get_char(@stream_or_alias, ?in_character)                  [ISO]
```

Examples

`get_char(Stream, Char).`
The contents of **Stream** are

```
'qwerty' ...
```

Char is unified with `'` (the atom containing just a single quote) and **Stream** is left as

```
qwerty' ...
```

`get_char(Stream, p).`
The contents of **Stream** are

```
qwerty ...
```

Fails and **Stream** is left as

```
werty ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`type_error(in_character, Char)`
Char is not a variable and it is not an in_character (a one-char atom or the atom **end_of_file**).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **get_char/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **get_char/1** the value of **S_or_a** is the current input stream.)

peek_char/1

peek_char/2

Description

`peek_char(Char)`

Is equivalent with

```
current_input(Stream), peek_char(Stream)
```

`peek_char(S_or_a, Char)`

Unifies the next char to be read from the text stream associated with **S_or_a** with **Char**. The next character is not removed from the source stream.

Template and modes

```
peek_char(?in_character)                                    [ISO]
peek_char(@stream_or_alias, ?in_character)                  [ISO]
```

Examples

`peek_char(Stream, Char).`
The contents of **Stream** are

```
'qwerty' ...
```

Char is unified with `'` (the atom containing just a single quote) and **Stream** is left unchanged:

```
'qwerty' ...
```

`get_char(Stream, p).`
The contents of **Stream** are

```
qwerty ...
```

Fails and **Stream** is left unchanged

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`type_error(in_character, Char)`
Char is not a variable and it is not an in_character (a one-char atom or the atom **end_of_file**).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **peek_char/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **peek_char/1** the value of **S_or_a** is the current input stream.)

put_char/1

put_char/2

Description

`put_char(Char)`

Is equivalent with

```
current_output(Stream), put_char(Stream,Char)
```

`put_char(S_or_a, Char)`

Outputs the character **Char** to the text stream associated with stream or alias **S_or_a**.

Template and modes

```
put_char(@character)                                    [ISO]
put_char(@stream_or_alias, @character)                  [ISO]
```

Examples

```
put_char(Stream,t).
```

If the stream indicated by **Stream** contains

```
... qwer
```

Succeeds and leaves that stream

```
... qwert
```

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Char is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`type_error(character, Char)`
Char is not a variable and it is not a character (a one-char atom).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **put_char/1** the value of **S_or_a** is the current input stream.)

nl/0

nl/1

Description

`nl`

Is equivalent with

```
current_output(Stream), nl(Stream)
```

`nl(S_or_a)`

Causes the current line on the text stream associated with stream or alias **S_or_a** to be terminated. (It is equivalent with **put_char(S_or_a, '\n')**).

Template and modes

```
nl                                                      [ISO]
nl(@stream_or_alias)                                    [ISO]
```

Examples

```
nl(st), put_char(st, a).
```

If the stream indicated by **st** contains

```
... qwer
```

the goal-sequence above succeeds and leaves that stream:

```
... qwer
a
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **nl/0** the value of **S_or_a** is the current input stream.)

get_code/1

get_code/2

Description

`get_code(Code)`

Is equivalent with

```
current_input(Stream), get_code(Stream, Code)
```

`get_code(S_or_a, Code)`

Succeeds if **Code** unifies with the character code corresponding to the next character read from text stream **S_or_a**. Note that the next character is always removed from the source stream, even if the predicate fails.

Template and modes

```
get_code(?in_character_code)                                [ISO]
get_code(@stream_or_alias, ?in_character_code)              [ISO]
```

Examples

`get_code(Stream, Code).`
The contents of **`Stream`** are

```
'qwerty' ...
```

`Code` is unified with character code of `',` the number 39, and **`Stream`** is left as

```
qwerty' ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`type_error(integer, Code)`
Code is not a variable and it is not an integer.
`representation_error(in_character_code)`
Code is an integer but not an in_character_code (a number between -1 and 255).

`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **get_code/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **get_code/1** the value of **S_or_a** is the current input stream.)

peek_code/1

peek_code/2

Description

`peek_code(Code)`

Is equivalent with

```
current_input(Stream), peek_code(Stream, Code)
```

`peek_code(S_or_a, Code)`

Succeeds if **Code** unifies with the character code corresponding to the next character to be read from the text stream **S_or_a**. The stream remains unchanged.

Template and modes

```
peek_code(?in_character_code)                                   [ISO]
peek_code(@stream_or_alias, ?in_character_code)                 [ISO]
```

Examples

`peek_code(Stream, Code), peek_code(Stream,Code).`
The contents of **`Stream`** are

```
'qwerty' ...
```

`Code` is unified with character code of `',` the number 39, and **`Stream`** is left unchanged.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`type_error(integer, Code)`
Code is not a variable and it is not an integer.
`representation_error(in_character_code)`
Code is an integer but not an in_character_code (a number between -1 and 255).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **peek_code/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **peek_code/1** the value of **S_or_a** is the current input stream.)

put_code/1

put_code/2

Description

`put_code(Code)`

Is equivalent with

```
current_output(Stream), put_code(Stream, Code)
```

`put_code(S_or_a, Code)`

Outputs the character of code **Code** to the text stream associated with stream or alias **S_or_a**.

Template and modes

```
put_code(@character_code)                                    [ISO]
put_code(@stream_or_alias, @character_code)                  [ISO]
```

Examples

`put_code(Str, 65).`
If the stream indicated by **`Str`** contains

```
... qwer
```

succeeds and leaves that stream

```
... qwerA
```

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Code is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`type_error(integer, Code)`
Code is neither a variable nor an integer.
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **put_code/1** the value of **S_or_a** is the current input stream.)
`representation_error(character_code)`
Code is an integer but it is not a character_code (a number between 0 and 255).

get_line/1

get_line/2

Description

`get_line(Line)`

Is equivalent with

```
current_input(Stream), get_line(Stream, Line)
```

`get_line(S_or_a, Line)`

Reads characters from text stream **S_or_a** until the next line-end ('\n') character. The atom formed from these characters is unified with **Line**. The line-end character will not be the part of the atom, but it will be read in.

Template and modes

```
get_line(?atom)
get_line(@stream_or_alias, ?atom)
```

Examples

```
get_line(Stream, Line).
```

The contents of **Stream** are

```
   qwerty
abcd ...
```

Line is unified with atom **qwerty**, and **Stream** is left as

```
abcd ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`type_error(atom, Line)`
Line is not a variable and it is not an atom.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **get_line/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **get_line/1** the value of **S_or_a** is the current input stream.)
`representation_error(max_atom)`
The length of line to be read is greater then the maximum length of atoms.

get_atom/2

get_atom/3

Description

```
get_atom(Code, Atom)
```

Is equivalent with

```
current_input(Stream), get_atom(Stream, Code, Atom)
```

```
get_atom(S_or_a, Code, Atom)
```

Reads characters from text stream **S_or_a** until the nearest character that has code **Code**. The atom formed from these characters is unified with **Atom**. The terminating **Code** character will not be the part of the atom, but it will be read in (consumed). **Code** can also be given as the special atom **end_of_file**.

Template and modes

```
get_atom(@character_code, ?atom)
get_atom(@stream_or_alias, @character_code, ?atom)
```

Examples

`get_atom(Stream, 101, X).`
Reads characters until the nearest **e** character. If the contents of **Stream** are

```
qwerty
```

X is unified with atom **qw,** and **Stream** is left as

```
rty
```

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Code is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`type_error(atom, Atom)`
Atom is not a variable and it is not an atom.
`type_error(integer, Code)`
Code is not a variable and it is not an in_character (a one-character atom or the atom **end_of_file**).
`representation_error(in_character_code)`
Code is an integer but not an in_character_code (a number between -1 and 255).
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **get_atom/2** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **get_atom/2** the value of **S_or_a** is the current input stream.)
`representation_error(max_atom)`
The length of the character sequence to be read is greater then the maximum length of atoms.

get_byte/1

get_byte/2

Description

`get_byte(Byte)`

Is equivalent with

```
current_input(Stream), get_byte(Stream, Byte)
```

`get_byte(S_or_a, Byte)`

Succeeds if **Byte** unifies with the next byte read from binary stream **S_or_a**. Note that the next byte is always removed from the source stream, even if the predicate fails.

Template and modes

```
get_byte(?in_byte)                                    [ISO]
get_byte(@stream_or_alias, ?in_byte)                  [ISO]
```

Examples

`get_byte(Stream, Byte).`
The contents of **Stream** are

```
39 40 41 42 43 44 45 46 ...
```

Byte is unified with byte 39, and **Stream** is left as

```
40 41 42 43 44 45 46 ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`type_error(in_byte, Byte)`
Byte is not an in_byte (an integer between -1 and 255).
`permission_error(input, text_stream, S_or_a)`
S_or_a represents a text stream. (In case of **get_byte/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **get_byte/1** the value of **S_or_a** is the current input stream.)

peek_byte/1

peek_byte/2

Description

`peek_byte(Byte)`

Is equivalent with

```
current_input(Stream), peek_byte(Stream, Byte)
```

`peek_byte(S_or_a, Byte)`

Succeeds if **Byte** unifies with the byte corresponding to the next byte to be read from the binary stream **S_or_a**. The stream remains unchanged.

Template and modes

```
peek_byte(?in_byte)                                    [ISO]
peek_byte(@stream_or_alias, ?in_byte)                  [ISO]
```

Examples

```
peek_byte(Stream, Byte), peek_byte(Stream,Byte).
```

The contents of **Stream** are

```
39 40 41 42 43 44 45 46 ...
```

Byte is unified with the number 39, and **Stream** is left unchanged.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).

`type_error(in_byte, Byte)`
Byte is not an in_byte (an integer between -1 and 255).
`permission_error(input, text_stream, S_or_a)`
S_or_a represents a text stream. (In case of **peek_byte/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **peek_byte/1** the value of **S_or_a** is the current input stream.)

put_byte/1

put_byte/2

Description

`put_byte(Byte)`

Is equivalent with

```
current_output(Stream), put_byte(Stream, Byte)
```

`put_byte(S_or_a, Byte)`

Outputs the byte **Byte** to the binary stream associated with stream or alias **S_or_a**.

Template and modes

`put_byte(@character_byte)` [ISO]
`put_byte(@stream_or_alias, @byte)` [ISO]

Examples

`put_byte(Str, 65).`
If the stream indicated by **`Str`** contains

```
... 60 61 62 63 64
```

succeeds and leaves that stream

```
... 60 61 62 63 64 65
```

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Byte is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`type_error(byte, Byte)`
Byte is not a byte (an integer between 0 and 255).
`permission_error(output, text_stream, S_or_a)`
S_or_a represents a text stream. (In case of **put_byte/1** the value of **S_or_a** is the current input stream.)

read_term/2

read_term/3

Description

`read_term(Term, Options)`

Is equivalent with

```
current_input(Stream), read_term(Stream, Term, Options)
```

`read_term(S_or_a, Term, Options)`

Inputs a sequence of tokens from the text stream associated with stream or alias **S_or_a** until an **end token** has been read. The options in **Options** list affect the reading (see section 3.10). It is a syntax error if end of stream is reached before an **end token** is found. The sequence of tokens is then parsed as a Prolog term, which is then unified with **Term**. The **end token** is removed from the stream.

Template and modes

```
read_term(?term, +read_options_list)                          [ISO]
read_term(@stream_or_alias, ?term, +read_options_list)
                                                              [ISO]
```

Examples

```
read_term(Str, X, [variable_names(VNames)]).
```

The stream **Str** contains the following characters:

```
qwerty(Abcd) + Abcd   .   efgh ...
```

X is unified with term **qwerty(_1) + _1**, and **VNames** is unified with the term **[_1 = 'Abcd'].** The stream is left as

```
 efgh ...
```

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **read_term/2** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **read_term/2** the value of **S_or_a** is the current input stream.)
`type_error(list, Options)`
Options is neither a list nor a partial list.
`domain_error(read_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid read option.
`syntax_error(err_atom)`
One or more tokens were read, but they could not be parsed as a term using the current set of operator definitions.

read/1

read/2

Description

`read(Term)`

Is equivalent with

```
current_input(Stream), read(Stream, Term)
```

`read(S_or_a, Term)`

Inputs a sequence of tokens from the text stream associated with stream or alias **S_or_a** until an **end token** has been read. It is a syntax error if end of stream is reached before an **end token** is found. The sequence of tokens is then parsed as a Prolog term, which is then unified with **Term**. The **end token** is removed from the stream.

This predicate is equivalent with calling **read_term/2** or **read_term/3** with an empty read option list.

Template and modes

```
read(?term)                                                    [ISO]
read(@stream_or_alias, ?term)                                  [ISO]
```

Examples

```
read(Str, X).
```

The stream **Str** contains the following characters:

```
qwerty(abcd) + abcd   .   efgh ...
```

X is unified with term **qwerty(abcd) + abcd**. The stream is left as

```
 efgh ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **read/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **read/1** the value of **S_or_a** is the current input stream.)
`syntax_error(err_atom)`
One or more tokens were read, but they could not be parsed as a term using the current set of operator definitions.

tread_term/2

tread_term/3

Description

`tread_term(Term, Options)`

Is equivalent with

```
current_input(Stream), tread_term(Stream, Term, Options)
```

```
tread_term(S_or_a, Term, Options)
```

Inputs a sequence of tokens from the text stream associated with stream or alias **S_or_a** as long as they form a valid Prolog expression. This term is then unified with **Term**. The options in **Options** list affect the reading (see section 3.10).

Template and modes

```
tread_term(?term, +read_options_list)
tread_term(@stream_or_alias, ?term, +read_options_list)
```

Examples

```
tread_term(Str, X, [vars(Vars)]).
```

The stream **Str** contains the following characters:

```
qwerty(Abcd) + Abcd efgh ...
```

X is unified with term **qwerty(_1) + _1**, and **Vars** is unified with the term **[_1].** The stream is left as

```
efgh ...
```

(Supposing that the atom **efgh** is not an infix nor a postfix operator.)

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of t**read_term/2** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of t**read_term/2** the value of **S_or_a** is the current input stream.)
`type_error(list, Options)`
Options is neither a list nor a partial list.
`domain_error(read_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid read option.
`syntax_error(err_atom)`
One or more tokens were read, but they could not be parsed as a term using the current set of operator definitions.

tread/1

tread/2

Description

```
tread(Term)
```

Is equivalent with

```
current_input(Stream), tread_term(Stream, Term)
```

```
tread(S_or_a, Term)
```

Inputs a sequence of tokens from the text stream associated with stream or alias **S_or_a** as long as they form a valid Prolog expression. This term is then unified with **Term**.

Template and modes

```
tread(?term)
tread(+stream_or_alias, ?term)
```

Examples

```
tread(Str, X).
```

The stream **Str** contains the following characters:

```
qwerty(abcd-efgh) ijkl ...
```

X is unified with term **qwerty(abcd-efgh)**, The stream is left as

```
ijkl ...
```

(Supposing that the atom **ijkl** is not an infix nor a postfix operator.)

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **tread/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of **tread/1** the value of **S_or_a** is the current input stream.)
`syntax_error(err_atom)`
The characters that were read in could not be parsed as a term using the current set of operator definitions.

tread_token/1

tread_token/2

Description

`tread_token(Token)`

Equivalent with

```
current_input(Stream), tread_token(Stream, Token)
```

`tread_token(S_or_a,Token)`

Reads the next token from the text stream associated with **S_or_a**, and unifies it with **Token**.

Template and modes

```
tread_token(?constant)
tread_token(@stream_or_alias,?constant)
```

Examples

```
tread_token(Str, X).
```

The stream **Str** contains the following characters:

```
qwerty(abcd-efgh) ijkl ...
```

X is unified with atom **qwerty**, The stream is left as

```
(abcd-efgh) ijkl ...
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(input, stream, S_or_a)`
S_or_a is not open for input (it is an output stream).
`permission_error(input, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of t**read_token/1** the value of **S_or_a** is the current input stream.)
`permission_error(input, past_end_of_stream, S_or_a)`
The stream **S_or_a** is in state **past_end_of_stream** and **eof_action(error)** applies to this stream. (In case of t**read_token/1** the value of **S_or_a** is the current input stream.)

write_term/2

write_term/3

Description

`write_term(Term, Options)`

Is equivalent with

```
current_output(Stream), write_term(Stream, Term, Options)
```

`write_term(S_or_a, Term, Options)`

Outputs **Term** to the text stream associated with stream or alias **S_or_a** in a form that is defined by the write options list **Options**.

Template and modes

```
write_term(@term, @write_options_list)                    [ISO]
write_term(@stream_or_alias, @term, @write_options_list)
                                                          [ISO]
```

Examples

`write_term(Str, [1,2,3], []).`
outputs the characters

```
[1,2,3]
```

to the stream associated with **Str**.

`write_term(Str, [1,2,3], [ignore_ops(true)]).`
outputs the characters

```
.(1,.(2,.(3,[ ])))
```

to the stream associated with **Str**.

`write_term(Str, '1<2', [quoted(true)]).`
outputs the characters

```
'1<2'
```

to the stream associated with **Str**.

`write_term(Str, '$VAR'(1)+'$VAR'(26),[numbervars(true)]).`
outputs the characters

```
B+A1
```

to the stream associated with **Str**.

Notes

The output appearance of a variable (unless **numbervars(true)** option is standing) is a sequence of decimal digit and underscore characters beginning with an underscore. The digits depend on the location of the variable in the memory, which may change during a garbage collection. Therefore, the same variable in subsequent write operations can appear in different forms.

Errors

`instantiation_error`
S_or_a is a variable
`instantiation_error`
Options is a partial list, or an element of the **Options** list is a variable.
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **write_term/2** the value of **S_or_a** is the current input stream.)
`type_error(list, Options)`
Options is neither a list nor a partial list.
`domain_error(write_option, Option)`
Option, an element of the **Options** list, is neither a variable nor a valid read option.

write/1

write/2

Description

`write(Term)`

Is equivalent with

```
current_output(Stream), write(Stream, Term)
```

`write(S_or_a, Term)`

Outputs **Term** to the text stream associated with stream or alias **S_or_a** applying the

[numbervars(true)]

write options list.

Template and modes

```
write(@term)                                                    [ISO]
write(@stream_or_alias,@term)                                   [ISO]
```

Examples

`write(Str, [1,2,3]).`
outputs the characters

```
[1,2,3]
```

to the stream associated with **`Str`**.

`write(Str, '$VAR'(1)+'$VAR'(26)).`
outputs the characters

```
B+A1
```

to the stream associated with **`Str`**.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **write/1** the value of **S_or_a** is the current input stream.)

writeq/1

writeq/2

Description

`writeq(Term)`

Is equivalent with

```
current_output(Stream), writeq(Stream, Term)
```

`writeq(S_or_a, Term)`

Outputs **Term** to the text stream associated with stream or alias **S_or_a** applying the

[quoted(true), numbervars(true)]

write options list.

Template and modes

```
writeq(@term)                                                    [ISO]
writeq(@stream_or_alias,@term)                                   [ISO]
```

Examples

```
writeq(Str, foo+bar).
```

outputs the characters

```
foo+bar
```

to the stream associated with **Str**.

```
writeq(Str, 'foo bar').
```

outputs the characters

```
'foo bar'
```

to the stream associated with **Str**.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **writeq/1** the value of **S_or_a** is the current input stream.)

write_canonical/1

write_canonical/2

Description

`write_canonical(Term)`

Is equivalent with

```
current_output(Stream), write_canonical(Stream, Term)
```

`write_canonical(S_or_a, Term)`

Outputs **Term** to the text stream associated with stream or alias **S_or_a** applying the

[ignore_ops(true), numbervars(true)]

write options list.

Template and modes

```
write_canonical(@term)                                    [ISO]
write_canonical(@stream_or_alias,@term)                   [ISO]
```

Examples

`write_canonical(Str, foo+bar).`
outputs the characters

```
+(foo,bar)
```

to the stream associated with **`Str`**.

`write_canonical(Str, [1,2,3]).`
outputs the characters

```
.(1,.(2,.(3,[])))
```

to the stream associated with **`Str`**.

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **write_canonical/1** the value of **S_or_a** is the current input stream.)

format/2

format/3

Description

`format(Ctrl_String,Arglist)`

Is equivalent with

```
current_output(Str), format(Str, Ctrl_String, Arglist)
```

`format(Stream_or_alias,Ctrl_String,Arglist)`

Description

The predicate writes out **Arglist** to the text stream associated with **Stream_or_alias** according to the format specification contained in the atom **Ctrl_String**. This predicate is non-standard, but it is modeled after, and essentially compatible with, Quintus Prolog (and with SICStus Prolog). It is a Prolog interface to the POSIX standard printf function family in C.

Arglist is a list of items to be written. If there is only one item it may be given as a term. But if this only element is itself a list then this simplification cannot be used. If there are no items then **Arglist** should be an empty list.

The default action on a format character in **Ctrl_String** is to print it. The character **~** (tilde) introduces a control sequence. A control sequence has the general form **~KC**. The character **C** determines the type of the control sequence. **K** is an optional integer parameter. If **K** is a ***** (asterisk) character, it means that the next argument in **Arglist** should be used as the numeric parameter of the control sequence. Not all control sequences accept the optional numeric parameter.

There are two types of control sequences: those that take an item (argument) from **Arglist** and those that do not. The length of **Arglist** must exactly match the number of argument-taking control sequences in **Ctrl_String**. The control sequences available are listed below.

The following control sequences take an argument from **Arglist**:

~a
: The argument should be an atom; it is written out without quoting.

~Kd
: The argument should be an integer; it is written out. If **K** is omitted, or it is zero, no decimal point is written. If **K** is positive, it indicates that a decimal point should be written inside the number and there should be **K** digits after the decimal point.

~KD
: Same as **~Kd** except that a comma will separate groups of three digits to the left of the decimal point.

~Kf
: The argument should be a float number; it is converted to decimal notation in the style **ddd.ddd** (**d** stands for a digit) where **K** is the number of digits after the decimal point. The default value for **K** is 6. If **K** is zero, no decimal point is written (the value will have the form of an integer, but it may be outside the valid integer range).

~Ke
~KE
: The argument should be a float; it is written out in exponential - scientific - form. It is converted in the style **d.dddEdd** where there is one digit before the decimal point and **K** is the number of digits after it. The default for **K** is 6. The first version writes **e** before the exponent, the second one writes **E**. **K** = 0 is interpreted as **K** = 1 (to enforce correct Prolog syntax).

~Kg
~KG
: The argument should be a float. It is printed either in decimal or in exponential notation. The style used depends on the value converted: exponential style will be used only if the exponent resulting from the conversion is less than -4 or greater than **K**. If exponential style is used, the exponent is introduced by **e** for the **g** format code and by **E** for the **G** code. K determines the number of significant digits shown: the displayed value will be rounded at the K-th digit. Non-significant zero digits after the decimal point are not shown, except the first one. An extra zero is appended if otherwise the displayed number would end with a decimal point. **K** = 0 is interpreted as **K** = 1 (to enforce correct Prolog syntax).

~Kc
: The argument should be a number. It is interpreted as a character code. The character is printed out **K** times. The default for **K** is 1.

~Kr
: The argument should be a number. **K** is interpreted as a radix. **K** should be between 2 and 36. The argument is written using letters **a**..**z** for digits larger than 9.

~KR
: The same as **~Kr**, except that the letters **A**..**Z** are used for digits larger than 9.

~Ks
: The argument should be a list of character codes. Exactly **K** characters will be written out. The default for **K** is the length of the list. If K is greater than the length of the list, padding space characters are appended.

~i
: The argument is ignored.

~w
: The argument is written out as by **write/1**.

~k
: The argument is written out as by **write_canonical/1**.

~q
: The argument is written out as by **writeq/1**.

The remaining control sequences take no argument form **Arglist** (except for K, if it is specified as '*').

~~
: Writes out the **~** (tilde) character.

~Kn
: Writes out **K** newline characters. **K** defaults to 1.

~N
: Writes out a newline if the current position is not already at the beginning of a line.

The following control sequences set column boundaries and specify padding. A column is defined as the available space between two consecutive column boundaries on the same line. A boundary is initially assumed at line position 0. The specifications only apply to the line currently being written.

When a column boundary is set ('~|' or '~+') and there are fewer characters written in the column just defined than its specified width, the remaining space is divided equally among the pad sequences ('~t') in the column. If there are no pad sequences, the column is space padded at the end.

If '~|' or '~+' specifies a position preceding the current position, the boundary is set at the current position.

~K|
: Sets a column boundary at line position **K**. **K** defaults to the current position.

~K+
: Sets a column boundary at **K** positions past the previous column boundary. **K** defaults to 8.

~Kt
: Specifies padding in a column. **K** is the fill character. The default for **K** is 32 (space). Any padding ('~t') after the last column boundary on a line is ignored.

Template and modes

```
format(+atom,@term)
format(@stream_or_alias,+atom,@term)
```

Examples

```
format(user_output,'Hello ~a~n',world).
```
Writes:
```
       Hello world (...and a newline)
```
```
format(user_output,'Hello ~1d ~w.',[23,[world]]).
```
Writes:
```
       Hello 2.3 [world].
```
```
format(user_output,'Hello~42tcruel~45t~|*world',[20]).
```
Writes:
```
       Hello*****cruel-----world
```

Errors

`instantiation_error`
S_or_a is a variable
`domain_error(stream_or_alias, S_or_a)`
S_or_a is not a variable and it is not a valid representation of a stream or a stream alias.
`existence_error(stream, S_or_a)`
S_or_a is not associated with an open stream.
`permission_error(output, stream, S_or_a)`
S_or_a is not open for output (it is an input stream).
`permission_error(output, binary_stream, S_or_a)`
S_or_a represents a binary stream. (In case of **format/2** the value of **S_or_a** is the current input stream.)
`instantiation_error`
Ctrl_String is a variable.
`type_error(atom, Ctrl_String)`
Ctrl_String is not an atom.
`type_error(integer, Arg)`
Argument **Arg** (an element of **Arglist**) should be an integer.
`type_error(atom, Arg)`
Argument **Arg** (an element of **Arglist**) should be an atom.
`type_error(float, Arg)`
Argument **Arg** (an element of **Arglist**) should be a float.
`domain_error(not_less_than_zero, Arg, [])`
Argument **Arg** (an element of **Arglist**) should be a positive integer.
`existence_error(term, Arglist)`
There are not enough arguments in the **Arglist** list.
`existence_error(control, Arglist)`
There are more arguments in the **Arglist** list then control sequences in **Ctrl_String**.
`representation_error(max_format)`
Internal buffer overflow.
`representation_error(character_code)`
The argument corresponding to a **'~Ks'** control sequence is not a valid character code list.
`evaluation_error(bad_format_item)`
There is a syntactically incorrect control sequence (introduced by a tilde character) in **Ctrl_String**. **Info_Oher** shows the position of the offending sequence within the format specification.

22. Operator and bracket handling

These predicates change, or give information about, the current operator and bracket set.

The operator and bracket sets are global; that is, there are no different sets for individual modules or processes. At the beginning of the execution only the predefined - built-in - operators and brackets are present in the CS-Prolog runtime system. That means that the different operator and bracket declarations in source files (used by the compiler) have no effect during runtime. The needed operators and brackets have to be created by **op/3** or **bracket/3** predicate.

op/3

Description

`op(Priority, Op_specifier, Operator)`

If **Operator** is an atom, then a new operator, named **Operator**, with specifier **Op_specifier** and priority **Priority** will be added to the system (or an old one will be modified accordingly).

If **Operator** is a (flat) list then the processing described above is performed for each element, beginning from the head. If an error occurs later, the result of the previous processing remains in effect.

If **Priority** equals 0, it means removing the property of **Operator** corresponding to **Op_specifier**. It is not considered an error if **Operator** does not posses the specified property at the time of the call.

Operator cannot be the `','` (comma) atom (so the priority of the predefined comma operator cannot be changed). An atom cannot have both infix and postfix operator specification at the same time.

There cannot be a left bracket and a prefix operator with the same name. There also cannot be a right bracket and an infix or postfix operator with the same name.

Template and modes

`op(@integer, @operator_specifier, @atom_or_atom_list)` `[ISO]`

Examples

`op(30, xfy, ++).`
Succeeds, and from now **++** is a right associative infix operator with priority 30.

`op(0, xfy, ++).`
Succeeds, and from now **++** is not an infix operator.

Errors

`instantiation_error`
Priority is a variable.

`instantiation_error`
Op_specifier is a variable.

`instantiation_error`
Operator is a partial list, or **Operator** is a list containing a variable.

`type_error(integer, Priority)`
Priority is neither a variable nor an integer.

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

`type_error(list,Operator)`
Operator is neither a partial list nor an atom nor a list.

`type_error(atom, Op)`
Op is a member of the list **Operator** and is neither a variable nor an atom.

`domain_error(operator_priority, Priority)`
Priority is an integer outside of the range [0,1200].

`domain_error(operator_specifier, Op_specifier)`
Op_specifier is an atom that is not a valid operator specifier.
`permission_error(modify, operator, ',')`
Operator is the comma atom or a list containing the comma.
`permission_error(create, operator, Operator)`
Op_specifier is a specifier such that Operator would have an invalid set of specifiers.

current_op/3

Description

`current_op(Priority, Op_specifier, Operator)`

Succeeds if **Operator** is an operator with properties defined by specifier **Op_specifier** and precedence **Priority**.

current_op/3 is resatisfiable. On backtracking, succeeds with all matching values.

The set of operators returned by a **current_op/3** call reflects the state in the moment of the call (logical view).

Template and modes

`current_op(?integer, ?operator_specifier, ?atom)` [ISO]

Examples

`current_op(P, xfy, OP).`
Succeeds three times if the predefined operators have not been altered. It unifies

`P with 1100, OP with ;`

and then

`P with 1050, OP with ->`

and then

`P with 1000, OP with ,`

Errors

`domain_error(operator_priority, Priority)`
Priority is neither a variable nor an integer inside of the range [0,1200].
`domain_error(operator_specifier, Op_specifier)`
Op_specifier is neither a variable nor a valid operator specifier atom.
`type_error(atom, Operator)`
Operator is neither a variable nor an atom.

bracket/3

Description

`bracket(Priority, Br_open, Br_close)`

A new bracket with open atom **Br_open**, close atom **Br_close** and priority **Priority** will be added to the system. The atom formed as the concatenation of **Br_open** and **Br_close** is called the name of the bracket.

If **Priority** equals 0, it requests the removal of the bracket property of the corresponding bracket defined earlier (if any).

Priority value 1202 also has a special meaning: the bracket so defined will behave as an alternative set of parentheses.

Br_open and **Br_close** must be different atoms, and neither of them can the empty string (`''`).

The priority of the predefined curly brackets cannot be changed.

It is an error if a new bracket definition is in conflict with an existing one, i.e., any one of **Br_open**, **Br_close**, or the name of the bracket is already involved in a currently existing bracket definition.

There cannot be a left bracket and a prefix operator with the same name. There also cannot be a right bracket and an infix or postfix operator with the same name.

These conflicts, however, are not signaled when deletion of a (non-existing) bracket is requested.

Template and modes

```
bracket(@integer, @atom, @atom)
```

Examples

```
bracket(300, <*, *>).
```
Succeeds, and from now a bracket with open atom **<*** and close atom ***>** can be used.

Errors

```
instantiation_error
```
Priority is a variable.

```
instantiation_error
```
Br_open is a variable.

```
instantiation_error
```
Br_close is a variable.

```
type_error(integer, Priority)
```
Priority is neither a variable nor an integer.

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

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

```
representation_error(max_atom)
```
The name of bracket (concatenation of **Br_open** and **Br_close**) cannot be represented because its size exceeds the limit for atoms.

```
domain_error(bracket_component_name, Priority)
```
Br_open or **Br_close** is the empty string atom (**''**).

```
domain_error(bracket_priority, Priority)
```
Priority is an integer outside of the range [0,1200] and is not the special value 1202.

```
consistency_error(identical_bracket_name_components, Br_open+Br_close)
```
Br_open and **Br_close** are identical (the same atom).

```
permission_error(modify, bracket, {})
```
An attempt to change the priority of the predefined curly brackets.

```
permission_error(create, bracket, Br_open+Br_close)
```
The requested bracket would be in conflict with a currently effective bracket definition (they have some common component), or with an operator definition (either **Br_open** is a prefix operator, or **Br_open** is an infix or postfix operator).

current_bracket/3

Description

```
current_bracket(Priority, Br_open, Br_close)
```

Succeeds if the arguments can be unified with the corresponding properties of a user bracket declaration currently in effect.

current_bracket/3 is resatisfiable. On backtracking, succeeds with all matching values.

The set of brackets returned by a **current_bracket/3** call reflects the state in the moment of the call (logical view).

Note that beside user brackets there is the system defined curly bracket pair (almost the same as the fixed user curly brackets, but using the non-quoted open-curly and close-curly tokens).

Template and modes

```
current_bracket(?integer, ?atom, ?atom)
```

Examples

```
current_bracket(P, <*, *>).
```

Succeeds if there is a bracket declaration in effect for these atoms.In this case **P** will be unified with the priority of that bracket.

Errors

```
domain_error(bracket_priority, Priority)
```

Priority is neither a variable nor an integer inside of the range [0,1202].

```
type_error(atom, Br_open)
```

Br_open is neither a variable nor an atom.

```
type_error(atom, Br_close)
```

Br_close is neither a variable nor an atom.

23. Atom processing

These predicates enable atoms to be processed as a sequence of characters or character codes. Facilities exist to split and join atoms, and to convert a single character to and from the corresponding character code.

atom_length/2

Description

`atom_length(Atom, Length)`

Unifies **Length** with the number of characters in **Atom**.

Template and modes

`atom_length(+atom, ?integer)` [ISO]

Examples

`atom_length('Hello world', N).`
Succeeds and unifies **N** with 11.

`atom_length('', N).`
Succeeds and unifies **N** with 0.

Errors

`instantiation_error`
Atom is a variable.
`type_error(atom, Atom)`
Atom is neither a variable nor an atom.
`type_error(integer, Length)`
Length is neither a variable nor an integer.
`domain_error(not_less_than_zero, Length)`
Length is an integer that is less than zero.

atom_concat/3

Description

`atom_concat(Atom_1, Atom_2, Atom_12)`

Unifies **Atom_12** with the concatenation of **Atom_1** and **Atom_2**.

atom_concat/3 is resatisfiable, but only when **Atom_12** is instantiated. On backtracking, successive values for **Atom_1** and **Atom_2** are generated.

Template and modes

`atom_concat(+atom, +atom, -atom)` [ISO]
`atom_concat(?atom, ?atom, +atom)` [ISO]

Examples

`atom_concat('Hello ', world, S).`
Succeeds and unifies **S** with 'Hello world'.

`atom_concat(T, world, helloworld).`
Succeeds and unifies **T** with **hello**.

```
atom_concat(T1, T2, hello).
```
Succeeds and unifies **T1** with the empty atom and **T2** with **hello**. On backtracking, **T1** becomes **h** and **T2** becomes **ello**, etc.

Errors

```
instantiation_error
```
Atom12 is a variable and one of **Atom1** and **Atom2** is also a variable.

```
type_error(atom, Atom1)
```
Atom1 is not a variable and it is not an atom.

```
type_error(atom, Atom2)
```
Atom2 is not a variable and it is not an atom.

```
type_error(atom, Atom12)
```
Atom12 is not a variable and it is not an atom.

```
representation_error(max_atom)
```
Atom12 cannot be represented because its size exceeds the limit for atoms.

sub_atom/5

Description

```
sub_atom(Atom, Before, Length, After, Sub_atom)
```

If **Atom** can be broken into three pieces in such a way that the first piece is **Before** characters long, the second piece is **Length** characters long, and the third piece is **After** characters long, then the second (middle) piece is unified with **Sub_atom**.

sub_atom/5 is resatisfiable. On backtracking all possible values for **Before**, **Length**, **After** and **Sub_atom** are generated.

Template and modes

```
sub_atom(+atom, ?integer, ?integer, ?atom)                    [ISO]
```

Examples

```
sub_atom('Hello world', 3, 2, _, S).
```
Succeeds and unifies **S** with **lo**.

```
sub_atom('Hello', _, 4, _, S).
```
Succeeds and unifies **S** with **Hell.** On backtracking, it succeeds once more unifying **S** with **ello**.

```
sub_atom(ab, Before, Length, After, Sub_atom).
```
Succeeds six times:

```
Before = 0, Length = 0, After = 2 Sub_atom = ''
Before = 0, Length = 1, After = 1 Sub_atom = a
Before = 0, Length = 2, After = 0 Sub_atom = ab
Before = 1, Length = 0, After = 1 Sub_atom = ''
Before = 1, Length = 1, After = 0 Sub_atom = b
Before = 2, Length = 0, After = 0 Sub_atom = ''
```

Errors

```
instantiation_error
```
Atom is a variable.

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

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

```
type_error(integer, Before)
```
Before is neither a variable nor an integer.

```
type_error(integer, Length)
```
Length is neither a variable nor an integer.

`type_error(integer, After)`
After is neither a variable nor an integer.
`domain_error(not_less_than_zero, Before)`
Before is an integer that is less than zero.
`domain_error(not_less_than_zero, Length)`
Length is an integer that is less than zero.
`domain_error(not_less_than_zero, After)`
After is an integer that is less than zero.

atom_chars/2

Description

`atom_chars(Atom, List)`

Unifies **List** with a list whose elements are the characters of **Atom**, or unifies **Atom** with an atom whose characters are the characters which are the elements of the list **List**.

Template and modes

```
atom_chars(+atom, ?character_list)                    [ISO]
atom_chars(-atom, +character_list)                    [ISO]
```

Examples

`atom_chars('', L).`
Succeeds and unifies **L** with **[].**
`atom_chars(hello, L).`
Succeeds and unifies **L** with **[h,e,l,l,o].**
`atom_chars(Str, [w,o,r,l,d]).`
Succeeds and unifies **Str** with **world**.

Errors

`instantiation_error`
Atom is a variable and **Chars** is a variable or a partial list or list containing a variable.
`type_error(atom, Atom)`
Atom is neither a variable nor an atom
`type_error(list, Chars)`
Chars is neither a partial nor a list.
`type_error(character, Char)`
Char, an element of the list **Chars**, is neither a variable nor a character (one-char atom).
`representation_error(max_atom)`
Atom cannot be represented because its size exceeds the limit for atoms.

atom_codes/2

Description

`atom_codes(Atom, List)`

Unifies **List** with a list whose elements are the character codes of the characters of **Atom**, or unifies **Atom** with an atom whose characters have a character code equal to the elements of the list **List**.

Template and modes

```
atom_codes(+atom, ?character_code_list)               [ISO]
atom_codes(-atom, +character_code_list)               [ISO]
```

Examples

`atom_codes('', L).`
Succeeds and unifies **L** with **[]**.

`atom_codes(hello, L).`
Succeeds and unifies **L** with [104,101,108,108,111].

`atom_codes(Str, [119,111,114,108,100]).`
Succeeds and unifies **Str** with **world**.

Errors

`instantiation_error`
Atom is a variable and **Codes** is a variable or a list containing a variable.
`type_error(atom, Atom)`
Atom is not an atom
`type_error(list, Codes)`
Codes is neither a partial list nor a list
`type_error(integer, Code)`
Code, an element of the list **Codes**, is neither a variable nor an integer.
`representation_error(character_code)`
Code, an element of the list **Codes**, is an integer but not a character code.
`representation_error(max_atom)`
Atom cannot be represented because its size exceeds the limit for atoms.

char_code/2

Description

`char_code(Char, Code)`

Unifies **Code** with character code for the character **Char** or unifies **Char** with an atom whose only character's code is **Code**.

Template and modes

```
char_code(+character, ?character_code)                    [ISO]
char_code(-character, +character_code)                    [ISO]
```

Examples

`char_code(a, Code).`
Succeeds and unifies **Code** with 97.

`char_code(Str, 0'c).`
Succeeds and unifies **Str** with **c**.

Errors

`instantiation_error`
Both **Char** and **Code** are variables.
`type_error(character, Char)`
Char is neither a variable nor a character (one-char atom).
`type_error(integer, Code)`
Code is neither a variable nor an integer.
`representation_error(character_code)`
Code is an integer but not a character code.

number_chars/2

Description

```
number_chars(Number, List)
```

Unifies **List** with a list whose elements are the characters that would be output if **Number** would be written out to a sink by **write_canonical/1** predicate, or unifies **Number** with a number which would be read from a source containing characters which are the elements of the list **List**.

Template and modes

```
number_chars(+number, ?character_list)                    [ISO]
number_chars(-number, +character_list)                    [ISO]
```

Examples

```
number_chars(1997, L).
```
Succeeds and unifies **L** with **['1','9','9','7'].**

```
atom_chars(N, ['2','0','0','0']).
```
Succeeds and unifies **N** with **2000.**

```
atom_chars(F, ['3','1','.','4','E','-','1']).
```
Succeeds and unifies **F** with **3.14**.

Errors

```
instantiation_error
```
Number is a variable and **List** is a variable, a partial list, or a list containing a variable.

```
type_error(number, Number)
```
Number is neither a variable nor a number

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

```
type_error(character, Char)
```
Char, an element of the list **List**, is neither a variable nor a character.

```
syntax_error(SyntErr)
```
The system could not read a number from the characters of **List**.

number_codes/2

Description

```
number_codes(Number, List)
```

Unifies **List** with a list whose elements are the character codes that would be output if **Number** would be written out to a sink by **write_canonical/1** predicate, or unifies **Number** with a number which would be read from a source containing characters whose codes are the elements of the list **List**.

Template and modes

```
number_codes(+number, ?character_codelist)                [ISO]
number_codes(-number, +character_codelist)                [ISO]
```

Examples

```
number_codes(1997, [49,57,57,55]).
```
Succeeds.

```
number_codes(2000.0, L).
```
Succeeds and unifies **L** with **[50,48,48,48,46,48]**.

```
atom_codes(Str, [0'1,0'.,0'0]).
```
Succeeds and unifies **Str** with **1.0**.

Errors

`instantiation_error`
Number is a variable and **List** is a variable, a partial list, or a list containing a variable.
`type_error(number, Number)`
Number is neither a variable nor a number
`type_error(list, List)`
List is neither a partial list nor a list.
`type_error(integer, Code)`
Code, an element of the list **Codes**, is neither a variable nor an integer.
`representation_error(character_code)`
Code, an element of the list **Codes** is an integer but not a character code.
`syntax_error(SyntErr)`
The system could not read a number from the character codes of **List**.

24. Date and Time

These predicates enable the user to get information about the current date, time, and about the elapsed CPU time. There is also some support for converting date and time representations and performing operations on them.

CS–Prolog II uses three different representations for date and time. The simplest representation for short time intervals is a non-negative Prolog integer expressing the length of the interval in hundredth of seconds. The time-handling predicates concerned will return only values for intervals shorter than one day (8,640,000). Some other predicates, for example **set_timeout/1**, accept only intervals shorter than half a day. Note that the CSP–II integer range can accommodate intervals over 7 days expressed in this unit.

The second representation is a multi–component structure for expressing full (UTC) date and time values. The structure is the following:

```
abs(YEAR,MONTH,DAY,HOUR,MIN,SEC,HUNDREDTH)
```

where the successive arguments are the year, month, day of month, hour, minute, second, and hundredths of second components of the expressed date and time. **YEAR** = 1 corresponds to year 1 AD, **YEAR** = 0 —to year 1 BC (year 0 UTC), etc. **MONTH** numbers begin from 1 for January; **DAY** expresses the day of month number (from 1).

The third representation is a — usually floating-point — number, expressing number of days. (The fractional part of the value represents the corresponding fraction of a day, e.g., the value 0.5 is half a day, or 12 hours, or 4,320,000 hundredth seconds.) This representation is the best suited for calculations involving dates or time intervals longer than one day. The precision of floating point numbers is sufficient for expressing intervals (or absolute dates) as large as 32,000 years with a rounding error less then one hundredth second.

Time intervals can also be interpreted as time points if a base time is assumed. CSP-II uses the time point midnight (0:0:0) UTC, January 1, year 2000 for this purpose. It is a good idea to maintain all time–point data relative to this basis internally; local form should be used only in connection with user interfaces. CSP–II predicates are adapted to this kind of usage. The epoch of this scale corresponds to Julian day number 2,451,544.5 (used in astronomy).

The reason behind shifting the origin by 2000 years with respect to UTC is that the resolution of floating point date–time values is not uniform; it is best around value 0. In the sequel, when speaking of CSP-II time-point values, we always assume this time coordinate system unless explicitly stated otherwise.

Note that calculations involving time–points can, in general, yield meaningful results only if the assumed time coordinate system is linear (which excludes systems with daylight saving) and only if all time points are expressed in the same coordinate system.

The (floating point) number–of–days and the (integer) hundredths–of–seconds representations can be converted into each other by simply multiplying and dividing, respectively (the scaling constant is 8,640,000). For example the following arithmetic expressions can be used:

```
HSECS is floor((ND - floor(ND)) * 8640000)
```

and

```
ND is HSECS * 8.64E6
```

CSP-II also provides predicates for converting values between the **abs(…)** structure representation (either local or UTC) and the floating point representation interpreted as point in time with the base point indicated above.

There are some limitations and peculiarities concerning these conversions. When converting dates to **abs(…)** format, the result is always is normalized, i.e., the components of the structure are within in their natural range (minutes, seconds between 0 and 59, hours between 0 and 23, months between 1 and 12, and days between 1 and the maximal day of that month). The inverse conversion, however, accepts unnormalized structures, too. The structure is interpreted component-by-component, from major to minor order. Each component is handled as a time interval relative to the time point designated by the previous part (or initially 0000-01-01 00:00:00 UTC). This treatment allows for simple arithmetic performed in different time units on the components of the structure. The range of dates accepted by the conversion predicate is rather liberal: from -5,884,323-05-15 to 5,874,898-06-03 as absolute dates, and the corresponding range from -2,149,935,193.0 to 2,145,032,102.0 as day numbers. For unnormalized input structures the range is somewhat narrower.

cpu_time/1

Description

```
cpu_time(Time)
```

Time must be a variable. The call unifies **Time** with the processor time consumed by the Prolog system from its start. The time is measured in hundredths of seconds, and wraps around after 24 hours (at the value 8,640,000).

In a multiprocessor implementation this time is related to the specific processor on which it is asked.

Template and modes

```
cpu_time(-integer)
```

Examples

```
cpu_time(X), format('Used ~2d sec.~n',[X]).
```

Writes out the time used by the program in seconds.

Errors

```
type_error(variable, Time)
```

The **Time** argument is not a variable.

wall_clock_time/1

Description

```
wall_clock_time(Time)
```

Time must be a variable. The call unifies **Time** with the value of the wall clock timer of the Prolog system, which is set to 0 when the program starts. The time is measured in hundredths of seconds, and wraps around after 24 hours (at the value 8,640,000).

Template and modes

```
wall_clock_time(-integer)
```

Examples

```
wall_clock_time(X), format('The program has been running for ~2d
seconds.~n',[X]).
```

Writes out the time elapsed since the program started, in seconds.

Errors

```
type_error(variable, Time)
```

The Time argument is not a variable.

calendar_time/1

Description

```
calendar_time(NDaysDateTime)
```

NDaysDateTime must be a variable. The call unifies **NDaysDateTime** with the floating point number-of-days time-point value corresponding to the current UTC time obtained from the operating system.

Template and modes

```
calendar_time(-float)
```

Examples

```
calendar_time(CT), H is 24 *(CT - floor(CT)),
    Hour is floor(H), Min is floor((H - Hour) * 60),
    format('~NThe time is now ~dh:~2min (GMT)~n',[Hour, Min]).
```

Writes out the UTC time of the day .

Errors

`type_error(variable, NDaysDateTime)`
The **NDaysDateTime** argument is not a variable.

abs_time/1

Description

`abs_time(StructDateTime)`

Unifies **SrtuctDateTime** with a structure

```
abs(YEAR,MONTH,DAY,HOUR,MIN,SEC,HUNDREDTH)
```

where the successive arguments are the year, month, day, hour, minute, second, and hundredths of second values of the current local date and time as perceived by the operating system.

Template and modes

`abs_time(?term)`

Examples

`abs_time(abs(Y,M,D,_,_,_,_)), format('~d/~d/~d',[Y,M,D]).`
Writes out the (local) date of execution.

`abs_time(abs(_,0,_,_,_,_,_)), format('~d/~d/~d',[Y,M,D]).`
Always fails because the predicate forms normalized structure components, where month number is between 1 and 12.

Errors

`domain_error(datetime_struct, StuctDateTime)`
StuctDateTime is neither a variable nor an **abs/7** datetime term.

gmtime_conversion/2

localtime_conversion/2

Description

`gmtime_conversion(NDaysDateTime, StuctDateTime)`

`localtime_conversion(NDaysDateTime, StuctDateTime)`

Performs conversion of date–time values between the number–of–days representation interpreted as point of time and the representation by the special structure

```
abs(YEAR,MONTH,DAY,HOUR,MIN,SEC,HUNDREDTH)
```

where the successive arguments are the year, month, day, hour, minute, second, and hundredths of second values of the date and time. At least one of the arguments must be (fully) instantiated. If **`NDaysDateTime`** is instantiated at the time of the call, then its value is converted to the structure representation and the result is unified with **`StuctDateTime`**. Otherwise the reverse conversion is performed (always resulting in floating point value). Note that in the latter case the **`abs(...)`** structure does not have to be normalized, i.e., values outside the normal range (even negative values) are accepted in the individual fields (see the introduction to this chapter).

The difference between the two predicates lies only in the interpretation of **StuctDateTime**. **gmtime_conversion/2** regards the fields of the structure as components of UTC date-time, while **localtime_conversion/2** uses the operating system's current local time zone setting when the structure is formed or interpreted (see also **localtime_info/4** and **time_zone/2**).

Template and modes

```
gmtime_conversion(+number, ?struct)
gmtime_conversion(?number, +struct)
localtime_conversion(+number, ?struct)
localtime_conversion(?number, +struct)
```

Examples

```
gmtime_conversion(ND, abs(1998,4,25,12,0,0,0)),
   format('~7f',[ND]).
```

Writes out **-615.5000000**, the number of days between the base date–time point (midnight of January 1, 2000) and the specified date-time (noon of April 25, 1998).

```
gmtime_conversion(-615.5, abs(1997,_,_,_,_,_,_)).
```

Fails because the timepoint specified by the floating point value is in year 1998 (see the previous example), not in 1997.

```
S = abs(1998,4,25,12,0,0,0), gmtime_conversion(GT, S),
    localtime_conversion(LT, S), Offs = truncate((GT - LT) * 86400),
    format(' The local time is offset by ~d seconds wrt. GMT.', [Offs]).
```

Writes out the signed difference, in seconds, between UTC and the local time being in effect on April 25, 1980.

Errors

```
instantiation_error
```

Both **NDaysDateTime** and **StuctDateTime** are variables.

```
type_error(number, NDaysDateTime)
```

NDaysDateTime is not a variable and not a number.

```
domain_error(datetime_struct, StuctDateTime)
```

NDaysDateTime is a variable, **StuctDateTime** is not a variable but it either is not an **abs/7** datetime term or the interpretation of its components results in a date outside the allowed range (for unnormalized components some theoretically allowed values may also be rejected).

```
domain_error(datetime_float, NDaysDateTime)
```

NDaysDateTime is a number but the date represented by it is outside the allowed date range.

gmtime_info/3

Description

```
gmtime_info(NDaysDateTime, DayOfWeek, WeekOfYear)
```

Provides additional information, not included in the structure representation of dates, about the date specified by time-point value NDaysDateTime interpreted as UTC date.

DayOfWeek is unified with the integer day of week code corresponding to that date. The day of week code is a small integer between 0 and 6; Sunday is coded as 0, Monday – 1, etc.

WeekOfYear is unified with a value in the range [0, 53] expressing the number of week inside the year (the first Monday in the year is the first day of week 1).

Template and modes

```
gmtime_info(+number, ?integer, ?integer)
```

Examples

```
gmtime_info(-615.5, 6, WN).
```

Succeeds because the timepoint corresponding to **-615.5000000** (noon of April 25, 1998) was on a Saturday; and unifies WN with 18, the number of that week within 1988.

Errors

`instantiation_error`
NDaysDateTime is an uninstantiated variable.
`type_error(number, NDaysDateTime)`
NDaysDateTime is neither a number nor a variable.
`type_error(integer, DayOfWeek)`
DayOfWeek is neither a variable nor an integer value.
`type_error(integer, WeekOfYear)`
WeekOfYear is neither a variable nor an integer value.
`domain_error(datetime_float, NDaysDateTime)`
NDaysDateTime is a number but the date represented by it is outside the allowed date range.

localtime_info/4

Description

`gmtime_info(NDaysDateTime, DayOfWeek, WeekOfYear, Daylight)`

Provides additional information, not included in the structure representation of dates, about the date specified by time-point value NDaysDateTime interpreted in the local time zone.

DayOfWeek is unified with the integer day of week code corresponding to that date. The day of week code is a small integer between 0 and 6; Sunday is coded as 0, Monday – 1, etc.

WeekOfYear is unified with a value in the range [0, 53] expressing the number of week inside the year (the first Monday in the year is the first day of week 1).

Daylight is unified with an indicator showing whether the alternate time zone (Daylight Saving Time) had been used for interpreting the date. Daylight = 1 means that DST is used; Daylight = 0 means that DST is not used because the specified time point does not fall inside that interval, and Daylight = -1 means that DST is not used because the system has no information in its database about the time point in question.

Note that the range of time point values handled by CSP-II is much wider then the corresponding range usually handled by the operating system. If the time-point specified falls outside the time interval accepted by the operating system then the primary time zone is used for the interpretation and Daylight is unified with -1.Otherwise the appropriate operating system service is used. The choice between the primary and the alternate time zones is somewhat arbitrary around change time (because of the discontinuity and the non-functional nature of the mapping). The ambiguity is resolved by the operating system by using specific rules.

Template and modes

`localtime_info(+number, ?integer, ?integer, ?integer)`

Examples

`gmtime_info(-615.5, 6, WN).`
Succeeds because the timepoint corresponding to **`-615.5000000`** (noon of April 25, 1998) was on a Saturday; and unifies WN with 18, the number of that week within 1988.

Errors

`instantiation_error`
NDaysDateTime is an uninstantiated variable.
`type_error(number, NDaysDateTime)`
NDaysDateTime is neither a number nor a variable.
`type_error(integer, DayOfWeek)`
DayOfWeek is neither a variable nor an integer value.
`type_error(integer, WeekOfYear)`
WeekOfYear is neither a variable nor an integer value.
`type_error(integer, Daylight)`
Daylight is neither a variable nor an integer value.

`domain_error(datetime_float, NDaysDateTime)`
NDaysDateTime is a number but the date represented by it is outside the allowed date range.

time_zone/2

Description

`time_zone(Primary, Alternate)`

The call unifies **Primary** with the description of the primary time zone currently used by the operating system for expressing local time. The description is a list of the form

```
[Name, Offset]
```

where **`Name`** is the symbolic name of the primary time zone as known by the operating system (as Prolog atom), and **`Offset`** is the time difference in number-of-days representation to be added to local times to obtain UTC time (this value being an interval is insensitive to the shift of origin).

If the operating system is currently set up to use an alternate time zone (for daylight saving correction) then **Alternate** is unified with the description of that alternate time zone in the same form as for **Primary**. Otherwise **Alternate** is unified with the empty list (**`nil`** atom, **`'[]'`**).

Note that the call does not give any indication about which time zone is in effect at the current time.

The details about time zones can be found in the operating system manuals.

Template and modes

`time_zone(-term, -term)`

Examples

```
time_zone(_, [_, _]).
```
Succeds if the operating system is currently aware of the existence of an alternate (daylight saving) time zone.

```
time_zone([Zone, Offs], _]), calendar_time(CT),
    LT is CT - Offs, H is 24 *(LT - floor(LT)),
    Hour is floor(H), Min is floor((H - Hour) * 60),
    format('~NLocal time is now ~dh:~dmin (~q)~n',[Hour, Min, Zone]).
```
Writes out the local time of the day assuming that the primary time zone is in effect.

Errors

`type_error(list, Primary)`
The **Primary** argument is neither a list nor a variable.

`type_error(list, Alternate)`
The **Alternate** argument is neither a list nor a variable.

localtime_atom/3

Description

`localtime_atom(NDaysDateTime, Format, DisplayString)`

Unifies **DisplayString** with an atom composed of the string representation of the local date-time value corresponding to the point-of-time specified by **NDaysDateTime**. The formatting of this string is controlled by **Format** (as specified for the POSIX C function **strftime**). The **Format** string consists of zero or more conversion specifiers and ordinary characters. All ordinary characters are copied unchanged into the result. Each conversion specifier is replaced by appropriate characters derived from the specified date value according to the following list:

```
%a     the locale's abbreviated weekday name.
%A     the locale's full weekday name.
%b     the locale's abbreviated month name.
%b     the locale's full month name.
%c     the locale's appropriate date and time representation.
%d     day of the month as a decimal number (1-31).
%H     hour (24-hour clock) as a decimal number (00-23).
%I     hour (12-hour clock) as a decimal number (00-11).
%j     day of the year as a decimal number (001-366).
%m     month as a decimal number (01-12).
%M     minute as a decimal number (00-59).
%p     the locale's equivalent of AM/PM designations associated with
       12-hour clock.
%S     second as a decimal number (00-61).
%U     the week number of the year (the first Sunday as the first day of
       week 1) as a decimal number (00-53).
%w     the weekday as a decimal number [0 (Sunday)-6).
%W     the week number of the year (the first Monday as the first day of
       week 1) as a decimal number (00-53).
%x     the locale's appropriate date representation.
%X     the locale's appropriate time representation.
%y     the year without century as a decimal number (00-99).
%y     the year with century as a decimal number.
%Z     the time zone name or abbreviation, or by no characters if no time
       zone is determinable.
%%     the percent sign character ('%').
```

Template and modes

`localtime_atom(+number, +atom, ?atom)`

Examples

```
calendar_time(CT),
    localtime_atom(DS, '%A, %B %d, %y  -  %I:%M %p (%Z)', DS), write(DS), nl.
```

Writes out something like **`Saturday, April 25, 1998  -  02:17 PM (MET DS)`**, if the locale is set to US English and the time zone is set to Middle European Time (DS indicates that Daylight Saving is in effect at that time).

Errors

`instantiation_error`
NDaysDateTime or **Format** is a variable.

`type_error(number, NDaysDateTime)`
NDaysDateTime is neither a number nor a variable.

`type_error(atom, Format)`
Format is not neither an atom nor a variable.

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

`domain_error(datetime_float, NDaysDateTime)`
NDaysDateTime is a number but the date represented by it is outside the allowed date range. If **ErrInfo-Other** is the atom **cannot_format_date**, then the year number in the result is outside the range 0000 - 9999, otherwise the general conversion range is involved (see the introduction to this chapter).

25. Prolog flags

A Prolog flag is an atom, which is associated with a value. Most of flags are defined by the CS-Prolog system and cannot be changed. These flags describe several values, properties that are characteristic for CS-Prolog. Some of flags are modifiable by the user; these alter the behavior of the CS-Prolog runtime system.

The following flags are present in CS-Prolog.

The main properties of the integer numbers:

`bounded`
: The atom **true** if the integers are always in the closed interval [**min_integer**..**max_integer**], otherwise the atom **false**.

`max_integer`
: The largest integer number.

`min_integer`
: The smallest integer number.

`integer_rounding_function`
: The atoms **down** or **toward_zero**.

The main properties of the float numbers:

`float_radix`
: The radix of float numbers.

`float_precision`
: The number of digits in the significand.

`float_min_exponent`
: The smallest exponent.

`float_max_exponent`
: The largest exponent.

`float_denorm`
: The atom **true** if denormalized numbers are included, otherwise the atom **false**.

`float_max`
: The largest float number.

`float_min_with_full_precision`
: The smallest positive float number which can be represented with full precision

`float_min`
: The smallest positive float number.

`float_epsilon`
: The maximum relative error in float values.

The main properties of Prolog terms:

`max_arity`
: The maximum arity allowed in compound terms.

General information about the runtime system:

`platform`
: A system dependent atom describing the host machine, e.g. **os2**, **unix**, **transputer** etc.

`version`
: An atom representing the version number of the runtime system e.g. '**1.1**'.

`number_of_processors`
: An integer, on monoprocessor systems it is always **1**.

Modifiable Prolog flags that alter the CS-Prolog runtime system:

`float_range_checking_function`
: One of the following atoms: **denormalize**, **underflow_exception_after_rounding**, **underflow_to_zero_after_rounding**. This flag influences arithmetic evaluation and reading terms from source streams.

`unknown`
: One of the following three atoms: **error**, **fail**, and **warning**. If an undefined predicate is called then the action of CS-Prolog depends on the value of this flag. The default is **error**.

`double_quotes`
: One of the following three atoms: **codes**, **chars**, and **atom**. Double quoted tokens will be read as a character code list, character list or quoted token, respectively, depending on the value of this prolog flag. The default is **codes**.

`garbage_collection`
: One of the atoms **on** or **off**. If set to off, garbage collection is not performed. The default is **on**.

`time_slice_length`
: An integer, the time slice length is measured in hundredth of seconds. The default is **200** (two seconds).

`discard_mttp`
: (discard mail to terminated processes) One of the atoms **on** or **off**. If set to **on**, events and interrupts routed to an already terminated process will be silently discarded. If set to **off**, the situation will be treated as overrun. See also **generate_event** and **cause_interrupt**. The default is **off**.

Modification of Prolog flags in multiprocessor environments can be costly, because the flags are global data to be known everywhere.

There are some modifiable Prolog flags that have no effect on the runtime system; they are reserved for future use:

`char_conversion`
: One of the atoms **on** or **off**.

`debug`
: One of the atoms **on** or **off**.

set_prolog_flag/2

Description

`set_prolog_flag(Flag, Value)`

If **Flag** is a Prolog flag and **Value** is a value within the range of values for **Flag**, then associates the value **Value** with the flag **Flag**.

Template and modes

`set_prolog_flag(@flag, @term)` `[ISO]`

Examples

`set_prolog_flag(undefined_predicate, fail).`
Succeeds, associating the value **`fail`** with flag **`undefined_predicate`**.

Errors

`instantiation_error`
Flag is a variable or **Value** is a variable.
`type_error(atom, Flag)`
Flag is not a variable and not an atom.
`domain_error(prolog_flag, Flag)`
Flag is not a prolog flag.
`domain_error(flag_value, Flag+Value)`
Value is not a valid value for **Flag**.
`permission_error(modify,flag, Flag)`
Value is appropriate for **Flag** but **Flag** is not modifiable.

get_prolog_flag/2

Description

```
get_prolog_flag(Flag, Value)
```

If **Flag** is a Prolog flag then unifies **Value** with the value currently associated with **Flag**.

Template and modes

```
get_prolog_flag(@flag, ?term)
```

Examples

```
get_prolog_flag(undefined_predicate, X).
```

Succeeds, unifying **X** with the value **`fail`**, **`error`**, or **`warning`** depending on the value of the flag **undefined_predicate**.

Errors

```
instantiation_error
```

Flag is a variable.

```
type_error(atom, Flag)
```

Flag is neither a variable nor an atom.

```
domain_error(prolog_flag, Flag)
```

Flag is an atom but it is not a prolog flag.

```
domain_error(flag_value, Flag+Value)
```

Value is neither a variable nor a valid value for **Flag**.

current_prolog_flag/2

Description

```
current_prolog_flag(Flag, Value)
```

Generates all **F** and **V** pairs that are Prolog flags and their associated values, and unifies **Flag** with **F** and **Value** with **V**.

current_prolog_flag/2 is resatisfiable. On backtrack it unifies all **F**, **V** pairs with **Flag** and **Value**. The used set of **F**, **V** values is frozen in the moment of the call. Therefore, if between two succeedings of the predicate, a flag is modified, this modification does not appear in output values.

Template and modes

```
current_prolog_flag(?flag, ?term)                    [ISO]
```

Examples

```
current_prolog_flag(X,on), write(X), nl, fail.
```

Writes out all Prolog flag names that have the value **`on`**.

Errors

```
type_error(atom, Flag)
```

Flag is neither a variable nor an atom.

```
domain_error(prolog_flag, Flag)
```

Flag is an atom but it is not a prolog flag.

26. Control predicates

These predicates provide additional facilities for affecting the control flow during execution. (see also Control constructs, section 1.2.3)

true/0

fail/0

Description

`true`

Succeeds

`fail`

Fails

Template and modes

`true` [ISO]
`fail` [ISO]

Errors

None

call/1

Description

`call(Term)`

Invokes (meta-calls) the term **Term**. If **Term** has a module prefix, the call is interpreted in the specified module, otherwise in the current module.

Template and modes

`call(+callable_term)` [ISO]

Examples

```
call(write('Hello')).
```
Writes out **`Hello`**.

Errors

`instantiation_error`
Term, its module prefix part or its call part is a variable.
`type_error(callable, Term)`
Term is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Term** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Term** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the functor of call and there is no predicate defined for this functor.

call_anywhere/1

Description

`call_anywhere(Term)`

Looks for the first module where the invocation (metacall) of the term **Term** is valid, and calls it there. **Term** should not contain a module prefix.

Template and modes

`call_anywhere(+callable_term)`

Examples

`call_anywhere(my_pred).`
Looks for a module where **`My_pred/0`** predicate is defined and visible from the current module and calls the first such procedure found.

Errors

`instantiation_error(call_anywhere(Term), 1, [])`
Term is a variable.
`type_error(callable, Term)`
Term is neither a variable nor a callable term.
`existence_error(procedure, Func)`
No matching procedure could be found for **Term** in any module. **Func** is the functor of **Call**.

\+ /1

not/1

Description

`\+ Term`

`not(Term)`

These two predicates perform the same task: invoke **Term** as a goal and succeed only if **Term** fails. The predicates implement negation by failure rather than true negation. They are not resatisfiable. If **Term** has a module prefix, the call is interpreted in the specified module, otherwise in the current module.

Template and modes

`\+(+callable_term)` [ISO]
`not(+callable_term)`

\+ is a predefined prefix operator.

Examples

`\+ true.`
Fails
`not(4 = 5).`
Succeeds

Errors

`instantiation_error`
Term, or its module prefix part, or its call part is a variable.
`type_error(callable, Term)`
Term is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Term** is not an atom.

`existence_error(module, Mod)`
Mod extracted from **Term** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Term** and there is no predicate defined for this functor.

once/1

Description

`once(Term)`

Succeeds if **Term** succeeds. **once/1** behaves as **call/1**, but is not resatisfiable. If Term has a module prefix, the call is interpreted in the specified module, otherwise in the current module.

Template and modes

`once(+callable_term)` [ISO]

Examples

`once(true).`
Succeeds (the same as **`true/0`**).
`once(repeat).`
Succeeds, but only once.

Errors

`instantiation_error`
Term, its module prefix part or its call part is a variable.
`type_error(callable, Term)`
Term is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Term** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Term** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Term** and there is no predicate defined for this functor.

repeat/0

Description

`repeat`

Always succeeds, it is resatisfiable indefinitely forever on backtracking. (Obviously a cut can change this.)

Template and modes

`repeat` [ISO]

Examples

`repeat, write('hello '), fail.`
Writes

```
hello hello hello hello hello ...
```

infinitly.

Errors

None

halt/0

halt/1

Description

`halt`

Exits from the Prolog, returns to the operation system that invoked it.

`halt(Retcode)`

Exits from the Prolog, returns to the operation system that invoked it, passing the value of **Retcode** as a message.

Template and modes

```
halt                                                    [ISO]
halt(@integer)                                          [ISO]
```

Examples

```
halt(-1).
```
Terminates the CS-Prolog program setting the return code to -1.

Errors

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

garbage_collection/0

Description

`garbage_collection`

Invokes explicitly the garbage collection procedure of the Prolog system.

Template and modes

`garbage_collection`

Errors

None

27. Exception handling

The concept of exception handling is described in details in chapter 5.

catch/3

Description

```
catch(Goal, Catcher, Recovery)
```

Invokes **Goal** in a protected way. If an error is signaled during the execution of the **Goal**, the control returns to the **catch/3** predicate. **Catcher** is unified with the error term and if the unification succeeds then the **Recovery** call is executed. If the unification fails then the exception cannot be handled here, and it is passed to the next protection level.

Template and modes

```
catch(+callable_term, ?term, +callable_term)                    [ISO]
```

Examples

```
catch(Goal, Y, true).
```

This call protects the **Goal** call from every error. If during the evaluation of **Goal** an exception is raised, the execution will continue with successful termination of the **catch/3** call.

Errors

`instantiation_error`
Recovery, its module prefix part, or its call part is a variable.
`type_error(callable, Recovery)`
Recovery is not a variable and not callable.
`type_error(atom, Mod)`
Mod extracted from **Recovery** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Recovery** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Recovery** and there is no predicate defined for this functor.

protected/3

Description

```
protected(Goal, Handler, Term)
```

Executes **Goal** and if during the execution an exception happens, the system will call the **Handler/5** predicate supplying it with five arguments as explained in section 4.6. If the handler succeeds then a substitution goal will be called instead of the call that caused the error or one of its ancestors. If the handler fails then the exception cannot be handled here, and it is passed to the next protection level.

Handler can contain a module prefix.

Template and modes

```
protected(+callable_term, +module_prefixed_atom, @term)
```

Examples

See section 4.8.

Errors

`instantiation_error`
Handler or its module prefix is a variable.
`type_error(atom, H)`
H is the remaining of **Handler** after removing the optional module prefix and it is neither a variable nor an atom.
`type_error(module, Mod)`
Mod extracted from **Handler** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Handler** is not a module name.
`existence_error(procedure, H/5)`
H/5 predicate indicator does not belong to a user-defined predicate, where **H** is the remaining of **Handler** after removing the optional module prefix.

throw/1

signal/1

Description

`throw(Term)`

`signal(Term)`

These two predicates perform the same task. They raise an exception, for which (a systematically renamed copy of) **Term** will be the error term. This exception can be handled in the same way as the system exceptions.

The additional **error info** term will be the default one (`[0, 0]`)

Template and modes

`throw(?term)` [ISO]
`signal(?term)`

Examples

```
catch(throw(apple(3)), apple(X), write(X)).
```
Succeeds writing out the number 3.

Errors

None

signal/2

Description

`signal(Term, Term2)`

The predicate raises an exception, **Term** will be the error term and **Term2** the additional **error info**. This exception can be handled in the same way as the system exceptions.

Template and modes

`signal(?term,?term)`

Examples

```
protected(signal(apple(3), cherry(4)), handler, 0).
```
The predicate **handler/5** will be called with the arguments of the **protected** call.

Errors

None

28. Solution collecting

When there are many solutions to a (sub)problem, and when those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

setof/3

Description

`setof(Term, Goal, Set)`

Set is unified with the list of all different instances of **Term** such that **Goal** succeeds. (These instances can be regarded as solutions of **Goal.**) The uninstantiated variables appearing in **Term** should not appear anywhere else in the clause except within the term **Goal**. The argument **Term** itself is used only as a template. It is possible for the collected instances to contain variables, but in this case the list **Set** will only provide an imperfect representation of what is in reality an infinite set.

If there are uninstantiated variables in **Goal** which do not also appear in **Term** (called free variables), then a call to this built-in predicate may backtrack generating alternative values for **Set** corresponding to different instantiations of the free variables of **Goal**. Two instantiations are different if they are not variants of each other, i.e. no renaming of variables can make them literally identical.

Variables occurring in **Goal** will not be treated as free if they are explicitly bound within **Goal** by an existential quantifier. An existential quantification is written:

```
Y ^ Q
```

meaning "there exists a **Y** such that **Q** is true", where **Y** is some Prolog variable or, more generally, a Prolog term containing variables. Those variables of **Q** are quantified by this expression which also appear in **Y**. **Q** itself also may be an existentially quantified goal expression, but the **^** operator is interpreted as existential quantifier only as long as it is the outermost (main) functor of the expression.

If there are no solutions of **Goal**, then the predicate fails. **Set** is never unified with the empty list.

The list **Set** is sorted in the standard order for terms and duplicate items are removed.

Template and modes

`setof(@term, +callable_term, ?list)` [ISO]

Examples

Suppose there are the following facts in the program:

```
parent(kate, mary).
parent(chris, mary).
parent(steve, emeric).
```

`setof(X, parent(X,Parent), L).`
Unifies **L** with **[chris, kate]** and **Parent** with **mary**. Later, after backtracking, unifies **L** with **[steve]** and **Parent** with **emeric**.

`setof(X, Parent^parent(X,Parent), L).`
Unifies **L** with **[chris, kate, steve]**.

`setof(p(X,Parent), parent(X,Parent), L).`
Unifies **L** with **[p(chris,mary), p(kate,mary), p(steve,emeric)]**.

Errors

`instantiation_error`
Goal, its module prefix part, or its call part is a variable.

`type_error(list, Set)`
Set is neither a partial list nor a list.

`type_error(callable, Term)`
Goal is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Goal** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Goal** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Goal** and there is no predicate defined for this functor.

bagof/3

Description

`bagof(Term, Goal, Bag)`

The operation of this predicate is the same as of **setof/3** except that the list (or alternative lists) unified with **Bag** will not be ordered, and may contain duplicates.

Template and modes

`bagof(@term, +callable_term, ?list)` [ISO]

Examples

Suppose there are the following facts in the program:

```
parent(kate, mary).
parent(chris, mary).
parent(steve, emeric).
```

`bagof(X, parent(X,Parent), L).`
Unifies **L** with **[kate, chris]** and **Parent** with **mary**. Later, after backtracking, unifies **L** with **[steve]** and **Parent** with **emeric**.

`bagof(X, Parent^parent(X,Parent), L).`
Unifies **L** with **[kate, chris, steve]**.

`bagof(p(X,Parent), parent(X,Parent), L).`
Unifies **L** with **[p(kate,mary), p(chris,mary), p(steve,emeric)]**.

Errors

`instantiation_error`
Goal, its module prefix part, or its call part is a variable.
`type_error(list, Bag)`
Bag is neither a partial list nor a list.
`type_error(callable, Term)`
Goal is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Goal** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Goal** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Goal** and there is no predicate defined for this functor.

findall/3

Description

`findall(Term, Goal, List)`

List is unified with the list of all instances of **Term** in all proofs of **Goal** found by Prolog. The order of the list corresponds to the order in which the proofs are found. The list may be empty and all variables in **Goal** are taken as existentially quantified. This means that each invocation of **findall/3** succeeds exactly once, and that no variables in **Goal** get bound (see **setof/3**).

Template and modes

```
findall(@term, @callable_term, ?list)                    [ISO]
```

Examples

Suppose there are the following facts in the program:

```
parent(kate, mary).
parent(chris, mary).
parent(sly, emeric).
```

```
findall(p(X,Y), parent(X,Y), L).
```

Collects all terms of form **p(X,Y)** for which the goal **parent(X,Y)** is true, so unifies **L** with list **[p(kate,mary),p(chris,mary),p(sly,emeric)]**.

Errors

`instantiation_error`
Goal, its module prefix part, or its call part is a variable.
`type_error(list, List)`
List is neither a partial list nor a list.
`type_error(callable, Term)`
Goal is neither a variable nor a callable term.
`type_error(atom, Mod)`
Mod extracted from **Goal** is not an atom.
`existence_error(module, Mod)`
Mod extracted from **Goal** is not a module name.
`existence_error(procedure, PredInd)`
PredInd is the predicate indicator extracted from **Goal** and there is no predicate defined for this functor.

29. Parallel programming built-in predicates

The following table lists all the involved built-in predicates. None of them is available in both the prelude and the working phase.

prelude phase

```
new/[2,3]
new_rt/[5,6]
kill/1
start_processes/0
```

working phase

```
open_channel_for_send/[1,2]
open_channel_for_receive/[1,2]
close_channel/1
send/2
test_send/[1,2]
receive/[2,3,4]
test_receive/[1,2]
deschedule_process/0
test_process/[1,2]
test_channel/2
process_list/1
channel_list/1
get_event/[1,2]
generate_event/[1,2]
cause_interrupt/2
```

All parallel-programming built-in predicates are non-backtrackable, i.e. the effect of them is not undone if the Prolog program performs backtrack over them.

The descriptions of the parallel-programming built-in predicates will refer for some common syntactical entities. They are defined bellow.

The testing predicates **test_channel/2**, **test_send/2** and **test_process/2** return some information about the global state of the system. One cannot fully rely on the answer because interrupts can change the state of some other processes, some external (from the requester's point of view) event changes the situation etc.

29.1 System-wide unique names

The names of the virtual processors, processes, channels and events are global for the whole CS-Prolog system. They are identified by system-wide unique names. These names are known everywhere in the program and any references to them denote the same entity. A system-wide unique name is an arbitrary Prolog term with two restrictions:

it can not be a list
it can not contain unbound variables

In the description of the template and mode sections if one of the above system-wide unique name is required then the following abbreviations will be used:

virproc	for virtual processor name
process	for process name
channel	for channel name
event	for event name

Note that Cs-Prolog provides some predefined identifiers for virtual processor names. They serve for identifying the physical processors under virtual processor names. They have the following form:

```
processor(N)
```

where N is an integer denoting the serial number of a physical processor. The serial numbers start at 1. Anywhere the virtual processor names can be used the above names do as well. The scheduler accepts them as virtual processor identifiers, but handles them as direct references to physical processors. In the multi-processor implementations of the Cs-Prolog system the assignment of the serial numbers to physical processor is defined in the Installation Guide of Cs-Prolog II.

29.2 Process goal

The creation of the CS-Prolog processes may require the determination of one or two executable goals The process goal can be any callable Prolog term. Note that if the goal contains unbound variables and if during its execution any of them is unified then the unification does not exceed the scope of the process.

In the description of the template and mode sections if process goal is required then the following abbreviation will be used:

callable_term	for process goal

29.3 Communication data

This definition covers three similar notions: the message, the event data and the interrupt data. The communication data can be an arbitrary Prolog term with a single restriction:

it must not be a single unbound variable

In the description of the template and mode sections if one of the above communication data is required then the following abbreviations will be used:

message	for message
data	for event and interrupt data

29.4 Channel specifier

Some built-in predicates operating on channels can accept more than one channel at a time. In the description of these built-in predicates the channel specifier is used.

The channel specifier may be one of the following three things:

a channel name (as defined above)
a non-empty Prolog list containing names of channels
an unbound variable

In the description of the template and mode sections if channel specifier is required then the following abbreviation will be used:

chanspec	for channel specifier

29.5 The deadlock signal

If during the execution of the program a global deadlock situation occurs then the CS-Prolog scheduler signals the following error in the main process:

```
system_error(Current_call,[deadlock])
```

where **Current_call** is the name of the predicate which is currently executed. Note that in this case only the main process is revitalized, in order to provide the program a last chance to perform some closing and error displaying operations (or to break the deadlock by some well-chosen action).

If the program has any chance to fall into deadlock then the user should take care of the correct handling of this error, otherwise this error will cancel the execution of the whole CS-Prolog program.

29.6 Error handling in real time processes

A memory resource error occurring when an event or interrupt is processed by the scheduler might cause the irrecoverable loss of that event or interrupt even if an exception handler tries to deal with the situation.

new/2

new/3

Description

```
new(Process_name, Process_goal)
new(Process_name, Process_goal, Processor_name)
```

Process_name and **Processor_name** are system-wide unique names. **Process_goal** should be a process goal. The **new/[2,3]** predicate creates a new process with **Process_name** as its name and **Process_goal** as its goal. The optional **Processor_name** serves for identifying the virtual processor to which the created process will be delegated. If the third argument is omitted then the target processor is undetermined and the CS-Prolog scheduler determines freely the target of the process. Note that the created process will start the execution of its goal only in the working phase, which begins with the invocation of the **start_processes/0** built-in predicate by the main process.

Template and modes

```
new(+process, +callable_term)
new(+process, +callable_term, +virproc)
```

Examples

```
new(my_new_process, goal_of_my_process).
```

A new process is created with the name **my_new_process** and in the working phase it will execute the **goal_of_my_process** predicate.

```
new(my_process(1), my_process_goal(1), same_processor),
new(my_process(2), my_process_goal(2), same_processor).
```

Two new processes are created for the same processor. Their goals differ only in the argument of the goal.

Errors

`permission_error(parallel, process, 0)`
The **new/[2,3]** built-in predicate has been called in the working phase

`instantiation_error`
The name of the process is a variable or contains a variable

`domain_error(unique_name, Process_name)`
The name of the process is not a valid unique name.

`instantiation_error`
The name of the virtual processor is a variable or contains a variable

`domain_error(unique_name, Processor_name)`
The name of the processor is not a valid unique name.

`permission_error(create, process, Process_name)`
A process with the same name has been already created.

new_rt/5

new_rt/6

Description

```
new_rt(Process_name, Event_handling_goal, Init_goal,
       Event_list, Periodicity)
new_rt(Process_name, Event_handling_goal, Init_goal,
       Event_list, Periodicity, Processor_name)
```

Process_name and **Processor_name** are system-wide unique names. **Event_handling_goal** `and` **Init_goal** should be process goals. The **Event_list** should be either an empty list or a list of system-wide unique names. The **Periodicity** should be one of the fixed terms detailed later.

The **new_rt/[5,6]** predicate creates a new real time process with **Process_name** as its name.

The initialization of the process is done by the execution of **Init_goal** when the real-time process becomes alive in the working phase. In normal cases this goal should succeed. If the execution of the **Init_goal** fails then the process will terminate (as failed) without the cyclic event-handling behavior being even started.

The incoming events are handled in the **Event_handling_goal**. When an event relevant to this real-time process occurs, this goal will be executed. It is the user's task to determine which event triggered the current execution of the real-time process and select the appropriate handler routine. The actual event can be inquired by the **get_event/[1,2]** built-in predicate. The **Event_handling_goal** should succeed unless the user wants the process to terminate. If this goal fails then the system terminates the execution of the process (indicating success). After termination of a real time process the events dedicated to it will be orphaned, i.e. the occurrence of such an event will cause the termination of the whole application due to missing handler.

The **Event_list** and the **Periodicity** arguments determine together the set of events for which the real-time process is triggered. The **Event_list** contains the event terms which can be triggered explicitly either by a process using the **generate_event/[1,2]** built-in predicate or by the environment of the CS-Prolog system using the foreign language interface.

The same event name must not be defined for more than one process. On the other hand, any process can generate any defined event.

The **Periodicity** argument determines which kind of implicitly generated timer event is to trigger the process. It can have one of the following three forms:

```
period(Time)
idle(Time)
no_event
```

Here **Time** is an integer describing a time interval in hundredth of second.

If **period(Time)** term is given then the real-time process will get periodically a timer event in every moment when the **Time** interval has been elapsed.

If **idle(Time)** term is given then the real-time process will get periodically a timer event in every moment when the **Time** interval has been elapsed and no other explicit event has occurred.

If **no_event** is given then no timer event will be generated at all.

The optional **Processor_name** serves for identifying the virtual processor to which the created process will be delegated. If the sixth argument is omitted then the target processor is undetermined and CS-Prolog scheduler determines freely the target of the process. Note that the created process will start the execution of its goal only in the working phase after the invocation of the **start_processes/0** built-in predicate.

It is important to note that the cyclic behavior of real time processes (executing the **Event_handling_goal** for each event occurrence) is organized by backtracking. The consequence of this is that side effects of backtrackable predicates (such as, for example, **assertz_b/1**) will be undone after the evaluation of the goal.

Template and modes

```
new_rt(+process, +callable_term, +callable_term, +event_list,
       +periodicity)
new_rt(+process, +callable_term, +callable_term, +event_list,
       +periodicity, +virproc)
```

Examples

```
new_rt(process_name, event_goal, init_goal,
        [event_name], no_event, processor_name).
```

This example creates a real-time process which executes the **`init_goal`** as initialisation and then it waits for the occurrences of the **`event_name`** event. Whenever such event occurs it executes the **`event_goal`**. This real-time process is not sensitive to the timed events.

```
new_rt(process_name, event_goal, true, [], period(300)).
```
This example creates a real-time process which has no initialisation goal (it executes the **true/0** built-in predicate). The real-time process executes in every 3 seconds the **event_goal**, but it does not wait for explicitly generated events.

```
new_rt(name, process_goal(event), process_goal(init),
        [ev(1), ev(2), stop], idle(700)).
```
This example creates a real-time process which has its event handling and initialisation goal as two different invocation of the same predicate. The real-time process is sensitive for three explicit events. If none of them event occurs for 7 seconds then a timed event is generated.

Errors

`permission_error(parallel, process, 0)`
The **new_rt/[5,6]** built-in predicate has been called in the working phase

`instantiation_error`
The name of the process is a variable or contains a variable.

`domain_error(unique_name, Process_name)`
The name of the process is not a valid unique name.

`instantiation_error`
The **Event_list** argument is a variable or contains a variable.

`domain_error(unique_name, Culprit)`
The **Culprit** element of the **Event_list** argument is not a valid unique name.

`permission_error(create, event, Culprit)`
The event name being specified (**Culprit** element of the **Event_list** argument) is either one of the following reserved system events: **period(_), idle(_), no_event,** in which case **ErrInfo-Other** will be the atom **reserved_event name**, or it has already occurred on the event list of this or another process. In the latter case **ErrInfo-Other** will be the atom **multiple_targets**.

`instantiation_error`
The **Periodicity** argument is a variable.

`type_error(periodicity, Periodicity)`
The **Periodicity** argument is not a valid periodicity term.

`type_error(integer, Periodicity)`
The **Periodicity** term contains non-integer argument.

`domain_error(positive_number, Periodicity_arg)`
The **Periodicity_arg** argument of **Periodicity** is an integer but it is not positive.

`instantiation_error`
The name of the virtual processor is a variable or contains a variable.

`domain_error(unique_name, Processor_name)`
The name of the virtual processor is not a valid unique name.

`permission_error(create, process, Process_name)`
A process with the same name has been already created.

kill/1

Description

`kill(Process_name)`

Process_name should be a system-wide unique name of an already existing process. the **kill/1** predicate removes the **Process_name** process created previously by a call of the **new/[2,3]** or **new_rt/[5,6]** predicate. Note that the **kill/1** predicate works only in the prelude phase. Consequently it can not be used for deleting running processes. Once the program is in the working phase, processes can only disappear when they finished the execution of their goal. This case has nothing to do with the **kill/1** built-in predicate.

If a running process has to be deleted in the working phase, an interrupt can be sent to it and the process should react to this interrupt by finishing execution.

The deletion of the main process is not allowed.

Template and modes

```
kill(+process)
```

Examples

```
kill(superfluous_process).
```
The **superfluous_process** is deleted, so it will not be started at the beginning of the working phase.

Errors

```
permission_error(parallel, process, 0)
```
The **kill/1** built-in predicate has been called in the working phase

```
instantiation_error
```
The name of the process is a variable or contains a variable

```
domain_error(unique_name, Process_name)
```
The name of the process is not a valid unique name.

```
existence_error(process, Process_name)
```
The process to be killed does not exist.

```
permission_error(remove, process, Process_name)
```
The main process would be deleted.

start_processes/0

Description

```
start_processes
```

The **start_processes/0** predicate ends the prelude phase and starts the working phase.

In those cases when the user does not want use the parallel feature of CS-Prolog the working phase may be empty. However, to achieve a more efficient memory handling, it is advisable to invoke a **start_processes/0** call at the beginning of the prelude phase. This makes the prelude phase almost empty and the single process will run entirely in the working phase. This technique is not obligatory, but the memory handling is more efficient this way.

This predicate can be invoked at most once and only by the main process.

Template and modes

```
start_processes
```

Examples

```
new(process_name, process_goal), start_processes.
```
This example creates a process and starts the working phase.

```
new(process_name, process_goal), start_processes, continue_main_goal.
```
This example is similar to the above one, but here the main process continues its task in the working phase too.

```
start_processes, main_goal_body.
```
This example does not create any process, but it instructs the CS-Prolog scheduler to regard the body of the main goal as the execution of a single process. (This technique increases the efficiency of the memory handling.)

Errors

```
permission_error(parallel, process, 0)
```
The **start_processes/0** built-in predicate has been called in the working phase.

open_channel_for_send/1

open_channel_for_send/2

open_channel_for_receive/1

open_channel_for_receive/2

Description

```
open_channel_for_send(Channel_name)
open_channel_for_send(Channel_name, Open_mode)
open_channel_for_receive(Channel_name)
open_channel_for_receive(Channel_name, Open_mode)
```

Channel_name should be a system-wide unique name. **Open_mode** is one of the fixed atoms detailed later.

The **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates open the appropriate end of the channel **Channel_name**. After the opening the caller process has the right to perform the appropriate operation on the channel until it is closed by a **close_channel/1** built-in predicate.

Open_mode (if it is present) must be one of the following three atoms:

```
unconditional
conditional
scheduled
```

Using them the behavior of the **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates can be modified. As far as the desired channel is instantly ready for being opened the execution of the predicate is identical in all of the three cases, the channel is opened successfully. In the case when the channel **Channel_name** cannot be currently opened in the appropriate direction (send or receive) because it is already opened by someone else, then **Open_mode** determines the behavior of the **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates as follows:

`unconditional`

The **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates signal an error. (Default case, identical the one argument predicate)

`conditional`

The **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates fail.

`scheduled`

The **open_channel_for_send/[1,2]** and **open_channel_for_receive/[1,2]** predicates cause the suspension of the caller process which remains suspended until the desired channel becomes ready to be opened in the appropriate direction (i.e. the current owner of the channel closes it). Then it succeeds as if the channel were ready at the moment of the invocation.

One process can not hold simultaneously both ends of a channel.

Template and modes

```
open_channel_for_send(+channel)
open_channel_for_send(+channel,+atom)
open_channel_for_receive(+channel)
open_channel_for_receive(+channel,+atom)
```

Examples

```
open_channel_for_send(channel_name).
```

This example opens the channel **`channel_name`** for sending

```
open_channel_for_send(channel_name, unconditional).
```

Same as the previous example.

```
open_channel_for_receive(channel(for(receive))),
                                    conditional).
```

If the **channel(for(receive))** is already opened for receive by someone else then this call will fail, otherwise succeed.

```
open_channel_for_receive(channel(0), scheduled).
```

This call will cause the caller process to be suspended if **channel(0)** is already opened by someone else, until it is released by a **close_channel/1** call. If **channel(0)** is instantly ready for opening this example succeeds without suspension.

Errors

In the description of errors the operation **open_channel_for_send/[1,2]** can always be substituted by **open_channel_for_receive/[1,2]**.

```
permission_error(parallel, process, 0)
```

The **open_channel_for_send/[1,2]** built-in predicate has been called in the prelude phase.

```
instantiation_error
```

The name of the channel is a variable or contains a variable or the **Open_mode** argument is a variable.

```
domain_error(unique_name, Channel_name)
```

The name of the channel is not a valid unique name.

```
type_error(atom, Open_mode)
```

The **Open_mode** argument is neither a variable nor an atom.

```
domain_error(channel_mode, Open_mode)
```

The **Open_mode** argument is an atom but not a valid channel open mode.

```
permission_error(open, channel, Channel_name)
```

The channel is already opened by someone else and the **unconditional** open mode was used.

```
permission_error(open, channel, Channel_name)
```

The channel is already opened by the caller process. The **ErrInfo-Other** will be the atom **already_open**.

```
permission_error(open, channel, Channel_name)
```

The channel is already opened in the reverse direction by the caller process. The **ErrInfo-Other** will be the atom **other_end_owned**.

close_channel/1

Description

```
close_channel(Channel_name)
```

The **Channel_name** should be a system-wide unique name. The **close_channel/1** predicate releases the end of a channel which has been previously opened by one of the **open_channel_for_send/[1,2]** or **open_channel_for_receive/[1,2]** calls. Only the owner of the channel can issue this call. After the closing other processes have the opportunity to open this channel again.

Template and modes

```
close_channel(+channel)
```

Examples

```
close_channel(channel_name).
```

The channel **channel_name** is closed.

Errors

```
permission_error(parallel, process, 0)
```

The **close_channel/1** built-in predicate has been called in the prelude phase.

```
instantiation_error
```

The name of the channel is a variable or contains a variable.

```
domain_error(unique_name, Channel_name)
```

The name of the channel is not a valid unique name.

`existence_error(channel, Channel_name)`
The channel **Channel_name** does not exist.
`permission_error(close, channel, Channel_name)`
The channel **Channel_name** is held by someone else, so the caller process has no right to close it.

send/2

Description

`send(Channel_name_or_list, Message)`

Channel_name_or_list should be a channel specifier. **Message** should be type of communication data. The **send/2** built-in predicate sends the message on the channel(s) determined by the **Channel_name_or_list** argument.

If **Channel_name_or_list** is a single channel then the channel must have been opened previously by an **open_channel_for_send/[1,2]** call. The scheduler tries to synchronize the sender and the receiver process. It means that if there is a process on the other end of the channel and the process is ready for receive a message on this channel (it is waiting in a **receive/[2,3,4]** call) the message passing takes place immediately and **send/2** succeeds. Otherwise, the caller process is suspended in the **send/2** call and will stay suspended until the receiver process is ready to accept the message. Then the message passing takes place and the **send/2** call succeeds.

If the **Channel_name_or_list** is a non-empty list of channel names then they all must have been opened correctly by previous **open_channel_for_send/[1,2]** calls. In this case **send/2** performs a broadcast-like message passing on the listed channels i.e. it sends the **Message** on every channel. The scheduler tries to establish the synchronism one by one for each channel. It means that if every process is ready to accept a message on the appropriate channel then the **Message** will be sent on every channel and **send/2** succeeds. If only a part of them is ready to accept a message (if any) then they will get the message, but **send/2** becomes suspended. As soon as a receiver process becomes ready to accept the message it will get it. The **send/2** call remains suspended until the last receiver process is ready to accept the message, and then **send/2** succeeds. Note that **send/2** does not require the global synchronism of all receiver processes with the sender process, instead, if a receiver process got its message it can continue the execution of its goal, regardless to fact that the sender process is already suspended because of other receiver processes. Furthermore, the order of channels in the **Channel_name_or_list** argument has nothing to do with the order as the messages are sent on channels. This order depends only on the ability of the receiver processes to accept.

If the **Channel_name_or_list** is an unbound variable then the set of all channels that the caller process opened for sending is assumed as the target of the **send/2** call. If the process has no channels opened for sending then a run-time error is signaled. The message passing takes place as if the list of all owned channels were given in the **Channel_name_or_list** argument. When at last **send/2** succeeds then the unbound variable in the **Channel_name_or_list** argument will be unified with the list of these channels.

If the **Message** argument contains unbound variables then during the message passing they will loose any relationship with variables in the sender process. If they are unified in the receiver process, this unification has no effect on the variables in the sender process at all.

Template and modes

```
send(?chanspec, +message)
```

Examples

```
send(single_channel, hello).
```
This example sends the atom **hello** on the **single_channel**.

```
send([channel(1), channel(2), other_channel],
      combined(message(1994, Variable))).
```
This example sends a more complex message on the three listed channels.

```
send(CHANNELS, for_everybody(hello)).
```
This example sends a message on all channels that the caller process owns. It is assumed that the **CHANNELS** variable is unbound before the call, but it will be unified with the list of the involved channels.

Errors

`permission_error(parallel, process, 0)`
The **send/2** built-in predicate has been called in the prelude phase.
`instantiation_error`
The name of the channel contains a variable.
`domain_error(unique_name, Channel_name)`
The name of the channel is not a valid unique name.
`existence_error(channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not owned by the caller process (may not exist at all).
`permission_error(output, channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not opened for sending.
`permission_error(output, channel, Culprit)`
The **Culprit** channel is a multiple element of the **Channel_name_or_list** list. The **ErrInfo-Other** will be the atom **repeated_occurrence**.
`instantiation_error`
The **Message** argument is a variable.

test_send/1

test_send/2

Description

`test_send(Channel_name_or_list)`
`test_send(Channel_name_or_list, Ready_channels)`

Channel_name_or_list should be a channel specifier. **Ready_channels** should be an unbound variable. The **test_send/[1,2]** built-in predicate serves for determining that in place of the **test_send/[1,2]** a **send/2** predicate with the same channel argument would be able to send at least one message immediately (there is at least one channel ready to accept a message) or not. If **test_send/[1,2]** succeeds it would, if fails it would not. The semantics of the **Channel_name_or_list** argument is identical to the one of the **send/2** predicate (single name, list of names or unbound variable for all owned channels). If the **Ready_channels** argument is given and **test_send/[1,2]** succeeds than it is unified with a list of the names of the channels through which a message would be able to be sent immediately. Depending on the current state of the involved channels this list may contain either a part of or the whole of the channel names specified in the first argument. Note that the order of the channel names in the returned list is undetermined.

Template and modes

```
test_send(?chanspec)
test_send(?chanspec, -chanspec)
```

Examples

Let's suppose that the program contains the following predicates:

```
optional_send :-
  test_send(my_channel), send(my_channel, my_message);
  true.

alternative_send :-
  test_send([left_channel, right_channel], Act_channel),
       (Act_channel = [left_channel], send(left_channel,
                                                 left_message;
        Act_channel = [right_channel], send(right_channel,
                                                  right_message);
        true);
  true.

send_to_all_now :-
```

```
  test_send(_,Ready_channels),
       send(Ready_channels,to_everybody).
```

`optional_send.`
This call will send a message on the channel **`my_channel`**, only if the message can be sent immediately.

`alternative_send.`
This call sends on one or none of the two channels a message depending on the fact that only one of them is ready to pass a message immediately or not.

`send_to_all_now.`
This call sends a message on every channels the caller process owns and ready to pass a message immediately.

Errors

`permission_error(parallel, process, 0)`
The **test_send/[1,2]** built-in predicate has been called in the prelude phase.

`instantiation_error`
The name of the channel contains a variable.

`domain_error(unique_name, Channel_name)`
The name of the channel is not a valid unique name.

`existence_error(channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not owned by the caller process (may not exist at all).

`permission_error(output, channel, Culprit)`
The **Culprit** (the **Channel_name_or_list** argument itself or one element of it) channel is not opened for sending.

`permission_error(output, channel, Culprit)`
The **Culprit** channel is a multiple member of the **Channel_name_or_list** list. The **ErrInfo-Other** will be the atom **repeated_occurrence**.

`type_error(variable, Ready_channels)`
The **Ready_channels** argument is a not variable.

receive/2

receive/3

receive/4

Description

```
receive(Channel_name_or_list, Variable)
receive(Channel_name_or_list, Variable, Winner_channel)
receive(Channel_name_or_list, Variable, Winner_channel,
        Remote_connection)
```

Channel_name_or_list should be a channel specifier. **Variable**, **Winner_channel**, and **Remote_connection** should be unbound variables. The **receive/[2,3,4]** built-in predicate receives a message from the channel (or one of the channels) specified by the **Channel_name_or_list** argument. The **receive/4** predicate is explained in more detail in the networking supplement.

If **Channel_name_or_list** is a single channel then the channel must have been opened previously by an **open_channel_for_receive/[1,2]** call. The scheduler tries to synchronize the receiver and the sender process. It means that if there is a process on the other end of the channel and the process is ready for send a message on this channel (it is waiting in a **send/2** call) the message passing takes place immediately, the message is unified with **Variable** and **receive/[2,3,4]** succeeds. Otherwise, the caller process is suspended in the **receive/[2,3,4]** call and will stay suspended until the sender process is ready to pass a message. Then the message exchange takes place, the message is unified with **Variable** and the **receive/[2,3,4]** call succeeds.

If the **Channel_name_or_list** is a non-empty list of channel names then they all must have been opened correctly by previous **open_channel_for_receive/[1,2]** calls. In this case **receive/[2,3]** performs exactly one message passing on one of them. The scheduler tries to establish the synchronism between the receiver process

and one of the sender processes. It means that if there is at least one sender process which can send a message then the scheduler chooses a process randomly among them, accepts its message, unifies it with the **Variable** argument and **receive/[2,3,4]** succeeds immediately. Otherwise, **receive/[2,3,4]** becomes suspended until at least one of the sender processes can yield a message, and then it accepts the message, unifies it with the **Variable** argument and **receive/[2,3,4]** succeeds. Note that sender processes that were able to send a message, but they were not chosen during this **receive/[2,3,4]** call, remain still suspended in their **send/2** calls and they will have a chance to send their message during the next occurrences of **receive/[2,3,4]** calls on the appropriate channels.

If the **Channel_name_or_list** is an unbound variable then the list of all channels that the caller process opened for receiving is assumed as the source of the **receive/[2,3,4]** call. If the process has no channels opened for receiving then a run-time error is signaled. The message passing takes place as if the list of all owned channels had been given explicitly in the **Channel_name_or_list** argument. When **receive/[2,3,4]** succeeds then the unbound variable in the **Channel_name_or_list** argument will be unified with the list of these channels.

If the **Winner_channel** argument is given (**receive/[3,4]**) then in case of success this argument will be unified with the name of the winner channel (i.e. the channel which was chosen to provide the message), regardless of whether **receive/[3,4]** was called with a channel specifier representing a single channel or multiple channels.

Note that starvation is possible for multi-way receive: the order of polling is fixed, and local partners (residing on the same processor) have better chance.

The fourth argument, **Remote_connection**, can be used for obtaining additional information about the sender when messages are being received from the telecommunication network. If the transfer was local, then this argument is unified with **nil**. Otherwise, when the transfer involved the network, the following list is unified with **Rem_Conn**:

```
[[ip_addr(Ipaddr), ip_port(Tcp_Port)], Connection_Name]
```

Ipaddr and **Tcp_port** give the full TCP/IP address of the remote application, and **Connection_Name** is the name of the sending remote connection. The net address is returned in **Remote_connection** because the sender application may be an unsolicited partner having no partner representation. If it is an explicit partner, its name can be retrieved using the **partner_current_attribute** predicate (see the networking supplement for more details).

Template and modes

```
receive(@chanspec, -term)
receive(@chanspec, -term, -channel)
receive(@chanspec, -term, -channel, -term)
```

Examples

```
receive(single_channel, Message).
```

This example waits a message to its **Message** variable on the **single_channel**.

```
receive([channel(1), channel(2), other_channel], Message,
         Winner).
```

This example waits one message on one of the three channels on whichever comes first. The message is unified with **Message**, the name of the winner channel is unified with **Winner**.

```
receive(CHANNELS, Messages).
```

This example waits one message on all channels that the caller process owns. It does not care which channel provided the message. It is assumed that the **CHANNELS** variable is unbound before the call, but it will be unified with the list of the involved channels.

```
receive(ch(air_port), Mess, _, Sender),
                      Sender = [_, airlink].
```

Receives a message and then checks whether it had been sent from a remote connection named **airlink**.

```
receive(ch(air_port), Mess, _, Sender),
      Sender = [[ip_addr(IA), ip_port(TP], _],
          partner_current_attribute(near, ip_addr, IA),
          partner_current_attribute(near, ip_port, TP).
```

Receives a message and checks whether it had been sent from a partner named **near**.

Errors

`permission_error(parallel, process, 0)`
The **receive/[2,3]** built-in predicate has been called in the prelude phase.
`instantiation_error`
The name of the channel contains a variable.
`domain_error(unique_name, Channel_name)`
The name of the channel is not a valid unique name.
`existence_error(channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not owned by the caller process (may not exist at all).
`permission_error(input, channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not opened for receiving.
`permission_error(input, channel, Culprit)`
The **Culprit** channel is a multiple member of the **Channel_name_or_list** list. The **ErrInfo-Other** will be the atom **repeated_occurrence**.
`type_error(variable, Variable)`
The **Variable** argument is not a variable.
`type_error(variable, Winner_channel)`
The **Winner_channel** argument is not a variable.
`type_error(variable, Rem_Conn)`
The **Rem_Conn** argument is not a variable.

test_receive/1

test_receive/2

Description

`test_receive(Channel_name_or_list)`

`test_receive(Channel_name_or_list, Ready_channels)`

Channel_name_or_list should be a channel specifier. **Ready_channels** should be an unbound variable. The **test_receive/[1,2]** built-in predicate serves for determining that in place of the **test_receive/[1,2]** a **receive/[2,3,4]** predicate with the same channel argument would be able to get at least one message immediately (there is at least one channel ready to send a message) or not. If **test_receive/[1,2]** succeeds it would, if fails it would not. The semantics of the **Channel_name_or_list** argument is identical to the one of the **receive/[2,3,4]** predicate (single name, list of names or unbound variable for all owned channels). If the **Ready_channels** argument is given and **test_receive/[1,2]** succeeds than it is unified with a list of the names of the channels through which a message would be able to be received immediately. Depending on the current state of the involved channels this list may contain either a part of or the whole of the channel names specified in the first argument. This argument is very similar to **Winner_channel** argument of the **receive/[2,3,4]** built-in predicate, but it is different because here the possible winners are gathered in a list. Obviously, in place of the **test_receive/[1,2]**, the **receive/[2,3,4]** built-in predicate would return only the name of the single winner channel. Note that the order of the channel names in the returned list is undetermined.

Template and modes

```
test_receive(?chanspec)
test_receive(?chanspec, -chanspec)
```

Examples

Let's suppose that the program contains the following predicates:

```
optional_receive :-
  test_receive(my_channel), receive(my_channel,
                                                My_message);
  true.

selective_receive :-
```

```
  test_receive([master_channel, slave_channel],
                   Act_channel),
       (Act_channel = [master_channel],
             receive(master_channel, Master_message);
        true);
  true.

receive_from_one_now :-
  test_receive(_,Ready_channels),
       receive(Ready_channels, Any_message).
```

`optional_receive.`
This call wants to accept a message on the channel **`my_channel`**, only if the message can be got immediately.

`selective_receive.`
This call accept a message on the **`master_channel`** only when this channel is the only one that is able to provide a message immediately.

`receive_from_one_now.`
This call tries to get one message on one of all channels the caller process owns and ready to yield a message immediately.

Errors

`permission_error(parallel, process, 0)`
The **test_receive/[1,2]** built-in predicate has been called in the prelude phase.

`instantiation_error`
The name of the channel contains a variable.

`domain_error(unique_name, Channel_name)`
The name of the channel is not a valid unique name.

`existence_error(channel, Culprit)`
The **Culprit** channel (the **Channel_name_or_list** argument itself or one element of it) is not owned by the caller process (may not exist at all).

`permission_error(input, channel, Culprit)`
The **Culprit** (the **Channel_name_or_list** argument itself or one element of it) channel is not opened for receiving.

`permission_error(input, channel, Culprit)`
The **Culprit** channel is a multiple member of the **Channel_name_or_list** list. The **ErrInfo-Other** will be the atom **repeated_occurrence**.

`type_error(variable, Ready_channels)`
The **Ready_channels** argument is not a variable.

deschedule_process/0

Description

`deschedule_process`

This predicate causes the active process to relinquish the remaining part of its current time slice in order to give way to other ready processes residing on the same physical processor.

The predicate can be used by a process performing some background activity for promoting the progress of some other processes.

Template and modes

`deschedule_process`

Errors

`permission_error(parallel, process, 0)`
The **deschedule_process/0** built-in predicate has been called in the prelude phase.

test_process/1

test_process/2

Description

```
test_process(Process_name)
test_process(Process_name, Process_state_record)
```

Process_name should be either a system-wide unique name or an unbound variable. In the latter case the name of the caller process is unified with **Process_name**. The **test_process/[1,2]** built-in predicate succeeds if there is a process in the system with **Process_name** as its name, otherwise fails. It unifies a process state record with **Process_state_record** argument (if it is required). The process state record is an eight-element Prolog list as follows:

```
[Process_name, Process_flag, Actual_processor,
        Owned_send_channels, Owned_receive_channels,
        Process_time, Event_queue_current_size,
        Event_queue_size_limit]
```

where

Process_name is the name of the tested process (it contains the same term as the first argument on exit).

Process_flag is one of the following atoms:

`active`
: the process currently is being executed;

`suspended`
: the process is ready to run, but the hosting processor is executing another process (waits for time slice);

`unsent`
: the process is waiting for the completion of a **send/[1,2]** call;

`unreceived`
: the process is waiting for the completion of a **receive/[2,3,4]** call;

`sendunopened`
: the process is waiting for the completion of an **open_channel_for_send/2** call ('scheduled' mode);

`receiveunopened`
: the process is waiting for the completion of an **open_channel_for_receive/2** call ('scheduled' mode);

`unevented`
: the (real-time) process is waiting for the next event with which to proceed (generated or timer event).

`terminated`
: the evaluation of the principal goal of the process is finished.

`short_term_transfer`
: the processor is waiting for the completion of some internal (service) message transfer that must complete without any further interaction with the user program.

Actual_processor is the symbolic name of the processor hosting the process, in the form

```
processor(N)
```

where **N** is 1 for the root processor, and consecutively increasing integer values are assigned to the other ('internal') processors.

Owned_send_channels is a Prolog list of the names of all the channels opened for sending (empty list if none).

Owned_receive_channels is a Prolog list of the names of all the channels opened for receiving (empty list if none).

Process_time is the processor time in hundredth of seconds used by the process.

Event_queue_current_size is the number of (generated) events waiting for being served by the process. Always zero for non-real-time processes.

Event_queue_size_limit is the current limit of the number of events waiting service at the process. Its value can be less than the current queue size if it had been lowered (events already on the queue are not discarded in that case). Always zero for non-real-time processes.

Note: In order to be prepared for further extension of the reported status components it is advised to treat the state record as a list containing **at least** eight elements.

Template and modes

```
test_process(?process)
test_process(?process, ?term)
```

Examples

`test_process(Act_process).`
This call unifies the name of the current process with **`Act_process`**.

`test_process(my_process, [_, State, _, _, _]).`
This call unifies the state of **`my_process`** with **`State`**.

`test_process(my_sender_process, [_, _, _, [], _]).`
This call checks whether the **`my_sender_process`** has channels opened for sending.

`test_process(_, [My_process|_]).`
This call returns the name of the currently executed process.

Errors

`permission_error(parallel, process, 0)`
The **test_process/[1,2]** built-in predicate has been called in the prelude phase.

`instantiation_error`
The name of the process contains a variable.

`domain_error(unique_name, Process_name)`
The name of the process is not a valid unique name.

test_channel/2

Description

`test_channel(Channel_name, Channel_state_record)`

Channel_name should be a system-wide unique name. The **test_channel/2** built-in predicate fails if there is no channel in the system with **Channel_name** as its name, otherwise it unifies the appropriate channel state record with the **Channel_state_record** argument. The channel state record is a five-element Prolog list as follows:

```
[Channel_name, Channel_flag,
        Sender_process, Receiver_process, Channel_state]
```

where

Channel_name is the name of the required channel (it is the same name as the first argument);

Channel_flag is one of the following atoms:

`sendopened`
: the channel is opened only for sending;

`receiveopened`
: the channel is opened only for receiving;

`opened`

the channel is opened for both sending and receiving;

Sender_process is one of the following:

nil (`[]`, the empty Prolog list) if the sending end of the channel is not opened;

the name of the process that opened the channel for sending, if it is presently opened by a process (by **open_channel_for_send/[1.2]**);

a list of form **`[port, PortName]`** if the sending end of the channel is presently associated with a networking Port object;

a list of form **`[dock, DockName]`** if the sending end of the channel is presently associated with a networking Dock object.

Receiver_process is one of the following:

nil (`[]`, the empty Prolog list) if the receiving end of the channel is not opened;

the name of the process that opened the channel for receiving, if it is presently opened by a process (by **open_channel_for_receive/[1.2]**);

a list of form **`[connection, ConnectionName]`** if the receiving end of the channel is presently associated with a networking Connection object.

Channel_state is one of the following four atoms:

`quiet`

no transfer operation is in progress;

`receiverequested`

the process at the receiving end of the channel requested a message, but the sending end didn't offer it yet;

`sendarrived`

the process at the sending end has offered a message, but the receiving end didn't claim it yet;

`ready_to_transfer`

both the sending and receiving ends of the channel are ready to transfer. Only channels connecting processes on different processors can be in this state

Networking is explained in detail in the companion Networking supplement of the manual.

Template and modes

```
test_channel(+channel, ?term)
```

Examples

```
test_channel(my_channel, [_, opened, _ ,_ ,_]).
```

This example tests whether **`my_channel`** is opened on both of its ends.

```
test_channel(remote_channel, [_, _, _, remote_process, _]).
```

This example checks whether the **`remote_channel`** is opened for receiving by the **`remote_process`**.

Errors

`permission_error(parallel, process, 0)`
The **test_channel/2** built-in predicate has been called in the prelude phase.

`instantiation_error`
The name of the channel is a variable or contains a variable.

`domain_error(unique_name, Channel_name)`
The name of the channel is not a valid unique name.

process_list/1

Description

```
process_list(Process_state_records)
```

The **process_list/1** predicate unifies a list of process state records with **Process_state_records** for each process on the actual processor. The process state records are the same as defined at the **test_process/[1,2]** built-in predicate.

Template and modes

```
process_list(?term)
```

Examples

```
process_list([[First_name|_]|_]).
```

This example unifies the name of the first process with **First_name**.

Errors

```
permission_error(parallel, process, 0)
```

The **process_list/1** built-in predicate has been called in the prelude phase.

channel_list/1

Description

```
channel_list(Channel_state_records)
```

The **channel_list/1** built-in predicate unifies a list of channel state records with **Channel_state_records** for each channel having at least one end at the actual processor. The channel state records are described at the **test_channel/2** built-in predicate.

Template and modes

```
channel_list(?term)
```

Examples

```
channel_list([First_name|_]|_]).
```

This example unifies the name of the first channel with **First_name**.

Errors

```
permission_error(parallel, process, 0)
```

The **channel_list/1** built-in predicate has been called in the prelude phase.

get_event/1

get_event/2

Description

```
get_event(Event_name)
get_event(Event_name, Event_data)
```

The **get_event/[1,2]** built-in predicate can be called only during the execution of an event-handling goal. It unifies **Event_name** with the name of the event that caused the execution of the event-handling goal. If the event contains data argument **Event_data** is unified with the data otherwise it is unified with an empty Prolog list. If an implicit timer event caused the restart of the event handling goal then the periodicity term given in the **Periodicity** argument of the **new_rt/[5,6]** built-in predicate is unified with **Event_name** and empty list

with **Event_data**. This predicate does not remove the event and its data from the event queue, so till the end of the current execution of the event handling goal, further invocations of the **get_event/[1,2]** built-in predicate will provide the same result. The event will be removed from the event queue by the system after the termination of the event-handling goal.

Template and modes

```
get_event(?variable)
get_event(?variable, ?variable)
```

Examples

The following examples show complete clauses, the head of which could appear as the **Event_handling_goal** argument of a **new_rt/[5,6]** call.

```
event_goal :-
  get_event(stop_event), !, fail;
  get_event(EVENT), write(EVENT).
```

If **stop_event** arrives then the real time process finishes its execution. For every other event the name of the event is written to the output.

```
event_goal :-
  get_event(event, stop_data), !, fail;
  get_event(event, Data), write(Data).
```

If an **event** arrives with the **stop_data** then the real time process finishes its execution. For every other occurrences of **event** the data of the event is written to the output.

Errors

`permission_error(parallel, process, 0)`
The **get_event/[1,2]** built-in predicate has been called in the prelude phase or during the execution other than the event-handling goal of a real-time process.

`permission_error(access, event, 0)`
The **get_event/[1,2]** built-in predicate has been called from the initializing goal (specified by the **Init_goal** argument of the corresponding **new_rt/[5,6]** call) of a real-time process.

generate_event/1

generate_event/2

Description

```
generate_event(Event_name)
generate_event(Event_name, Event_data)
```

The predicate serves for generating an explicit internal event. **Event_name** should be the (system-wide unique) name of an existing event kind (defined during real-time process creation). **Event_data** should be a communication data item (any term except a single unbound variable). If this argument is omitted then an empty list is assumed as the data item to be passed.

The **generate_event/[1,2]** call always succeeds if called with proper arguments.

The asynchronous nature of communication using events implies that a waiting queue is used to store those events having arrived while the process is still busy with serving previously arrived events. It is the user's responsibility to ensure that the service rate be higher than the arrival rate (perhaps through implementing some flow-control discipline). The queue merely serves to accommodate temporary surges in arrivals. The length of the queue is restricted by a modifiable value, which is set to 100 when the process starts, and can be changed through the call of **set_event_qsize_limit/[1,2]**.

If the event queue of a process is full and new event arrives to this process, a

```
system_error(event_queue_overrun, ...)
```

exception is raised (which will be directed to main_process), and the offending event is discarded. With the aim of reducing the frequency of such exceptions only the first overrun event is signaled from a possible sequence until the 'non-full' state of the queue is restored (by the process consuming events from the head of the queue).

If an event is generated when the servicing process of that event type has already terminated, the new event can be treated as overrun or just be silently discarded, under the control of the **discard_mttp** prolog flag.

Template and modes

```
generate_event(+event)
generate_event(+event, +data)
```

Examples

```
generate_event(my_event).
```
This example generates an occurrence of the **my_event**. If the real-time process asks the event data of that occurrence it will get an empty Prolog list as event data.

```
generate_event(my_event, something_occured).
```
This example generates an occurrence of the **my_event** with **something_occured** as event data.

Errors

```
permission_error(parallel, process, 0)
```
The **generate_event/[1,2]** built-in predicate has been called in the prelude phase.

```
instantiation_error
```
The name of the event is a variable or contains a variable.

```
domain_error(unique_name, Event_name)
```
The name of the event is not a valid unique name.

```
existence_error(event, Event_name, [])
```
The required event does not exist.

```
permission_error(access, event, Event_name)
```
Event_name is one of the following reserved system events: **period(_), idle(_), no_event**.

```
instantiation_error
```
The **Event_data** argument is a variable.

set_timeout/1

Description

```
set_timeout(Time)
```

The **Time** argument should be an integer representing a virtual time interval in hundredth of second, which must be shorter than half a day (4320000). The **set_timeout/1** built-in predicate initializes the alarm clock of the caller process and succeeds. The execution of the caller process continues. As soon as the time interval specified in the **Time** argument elapses the scheduler will signal the following interrupt, unless the alarm clock had been stopped by the **reset_timeout/0** built-in predicate in the meantime:

```
interrupt(task_timeout,[])
```

It is the user's responsibility to take care of the correct handling of this interrupt, otherwise this interrupt will cancel the execution of the whole CS-Prolog program.

The alarm clock is ticking only when the process is being executed, so it shows not the real time, but rather the wall-clock time that passed for the process.

The alarmed process is not interrupted immediately if the process is executing an I/O built-in predicate. It will be interrupted when the appropriate I/O operation is terminated.

Template and modes

```
set_timeout(+integer)
```

Examples

```
set_timeout(700).
```

This example will cause a timeout interrupt at the caller process after 7 seconds active time.

Errors

```
instantiation_error
```

The **Time** argument is a variable.

```
type_error(integer, Time)
```

The **Time** argument is not an integer.

```
domain_error(timeout, Time)
```

The **Time** argument is an integer outside the boundaries of the acceptable range **[0,4319999]**.

reset_timeout/0

Description

```
reset_timeout
```

The **reset_timeout/0** built-in predicate can be used to stop the alarm clock of the caller process before the time-out, previously set by the **set_timeout/1** built-in predicate, sets it off. The call always succeeds.

Template and mode

```
reset_timeout
```

Examples

Let's suppose that the program contains the following predicate:

```
set_reset :-
  set_timeout(1000), if_ok, reset_timeout;
  true.
```

```
set_reset.
```

This call succeeds if the **if_ok** predicate is executed successfully within 10 seconds, otherwise a timeout interrupt is generated.

Errors

None.

cause_interrupt/2

Description

```
cause_interrupt(Process_name, Interrupt_data)
```

Process_name should be a system-wide unique name and **Interrupt_data** should be a communication data. The **cause_interrupt/2** built-in predicate signals the following interrupt at the **Process_name** process:

```
interrupt(Current_predicate,program,Interrupt_data)
```

where **Current_predicate** is the predicate indicator of the procedure the execution of which had been interrupted, and the last argument of the interrupt term is the same term as the **Interrupt_data** argument of the **cause_interrupt/2** built-in predicate. Note that the execution of the interrupted process will be broken even if it is suspended for some reason. In this case the suspended call is abandoned.

It is the user's responsibility to take care of the correct handling of this interrupt, otherwise this interrupt will cancel the execution of the whole CS-Prolog program.

The target process is interrupted at the first possible interruption point encountered after the arrival of an interrupt request (generated by **cause_interrupt** or of any other kind).

Note that the majority of built-in predicates do not contain such interruption point inside. In particular the I/O predicates cannot be interrupted; this can be felt when inputting from the keyboard.

The most important exception from the above rule is the set of communication-oriented predicates that depend on the actions of other processes (**send**, **receive**, **open** with the **scheduled** option). These can be interrupted during wait.

A flow control scheme is used to prevent flooding the system with unserviced interrupts. It is similar to the scheme used in controlling the size of the event queue for real-time processes (see **generate_event/[1,2]**). If the interrupt queue of a process is full and new interrupt arrives to this process, a

```
system_error(interrupt_queue_overrun, ...)
```

exception is raised (which will be directed to the main process), and the offending interrupt is discarded. The size of interrupt queue is fixed (100 for main process, 10 for any other process).

The treatment of interrupts directed to an already terminated process is controlled by the prolog flag **discard_mttp**.

Real time tasks cannot be interrupted (at least in any sensible way), because part of the time is spent in the outer loop, where no handler can be installed. Anyway, they can be alerted by events.

If the addressed process is executing an exception handler when the interrupt is caused, the interrupt does not break the handler. The effect of the interrupt is deferred until the exception handling process is finished.

Template and modes

```
cause_interrupt(+process, +data)
```

Examples

```
cause_interrupt(other_process, my_interrupt).
```

This example signals an interrupt at the **other_process** and the last argument of the interrupt term will be **my_interrupt**.

Errors

```
permission_errorparallel, process, 0)
```

The **cause_interrupt/2** built-in predicate has been called in the prelude phase.

```
instantiation_error
```

The name of the process is a variable or contains a variable.

```
domain_error(unique_name, Process_name)
```

The name of the process is not a valid unique name.

```
existence_error(process, Process_name)
```

The required process does not exist.

```
instantiation_error
```

The **Interrupt_data** argument is a variable.

set_event_qsize_limit/1

set_event_qsize_limit/2

Description

```
set_event_qsize_limit(Size)
set_event_qsize_limit(Process_name, Size)
```

Process_name should be the (system-wide unique) name of an existing process (although the process may be in terminated state). It specifies the process that is the target of the operation. In the one-argument variant the target process is the current one (in which the call is executed).

The **Size** argument should be a positive integer. This value restricts the number of events (originated by **generate_event/[1,2]**) that can be put on the waiting queue of the process issuing the call.

The initial limit is 100 (when the process is started). If a new event arrives when the queue is already full (according to the current limit setting), the following exception is routed to main process:

```
system_error(event_queue_overrun, EventName).
```

and the new message is discarded. Any further events will be discarded without causing repeated exception as long as the queue remains **quiet**. Sensitivity for overrun is restored when the process consumes enough messages from the queue (by servicing them) so that the queue becomes non-full.

Note that if **Size** is less than the current number of actually waiting messages, the limit will be reset to the new value but no events will be discarded from the tail of the queue and overrun will not be signaled on account of the fact that the queue is overpopulated. Only new arrivals can raise the exception.

When a real-time process terminates, the queue size limit is set to 0, so the first event arriving after termination will raise the **overrun** exception. This default behavior can be changed by setting the **discard_mttp** prolog flag to **on**. In this case the late coming events (and interrupts) will be silently discarded.

For non-real-time process targets (which cannot receive events) and for terminated processes the predicate has no effect (but succeeds).

Notes

Timer events are treated in a specific way, they don't contribute to the queue length.

Events might be enqueued to a real-time process even before the initial goal of the process is started!

If a real-time process terminates while there are any waiting events and/or interrupts on its queues, these items are treated as latecomers (raise the corresponding signal or just are silently discarded under the control of **discard_mttp** flag). This behavior is different from the case when the queue size limit is lowered below current queue size, because in this situation it is sure that the waiting items cannot be serviced later.

Template and modes

```
set_event_qsize_limit(+integer)
set_event_qsize_limit(+process, +integer)
```

Errors

`permission_error(parallel, process, 0)`
The predicate was called in the prelude phase.

`instantiation_error`
The **Size** argument is an unbound variable.

`type_error(integer, Size)`
The **Size** argument is not an integer.

`domain_error(queue_size,Size, [])`
The **Size** argument is outside the boundaries of acceptable range **[1,65535]**

`existence_error(process, Process_name)`
Unknown target process.

30. Miscellaneous predicates

The predicates in this section perform diverse functions, not falling into the other thematic groups. Among others here are collected those predicates which access some operating system services.

current_environment/2

Description

`current_environment(Name, Value)`

Queries the environment variables maintained by the operating system for the application. Generates all **N** and **V** pairs (both atoms) that are the name of an environment variable and the associated value in the current environment; and unifies **Name** with **N**, **Value** with **V**. **current_environment/2** is resatisfiable: on backtrack it unifies all **N**, **V** pairs (collected and frozen at the time of the initial execution) with **Name** and **Value**. So if the environment is changed between two succeedings of the same call, the change is not reflected in the result.

Template and modes

`current_environment(?atom, ?atom)`

Examples

```
current_environment(Name, Value),
    format('~a ~33|= ~a~n',[Name,Value]), fail ; true.
```

Writes out the whole environment on the current ouput.

```
current_environment('PATH', Path), write(Path), nl.
```

Writes out the value of the **PATH** environment variable on the current ouput.

Errors

`type_error(atom, Name)`
Name is neither an atom nor a variable.
`type_error(atom, Value)`
Value is neither an atom nor a variable.

current_working_directory/1

Description

`current_working_directory(PathName)`

Unifies **PathName** with an atom composed from the string returned by the operating system as the absolute pathname of the current working directory.

Template and modes

`current_working_directory(?atom)`

Errors

`type_error(atom, PathName)`
PathName is neither an atom nor a variable.
`system_error`
The operating system indicated an error (probably insufficient access rights for some element of the path).
ErrInfo-Other contains the error number obtained from the operating system.

change_working_directory/1

change_working_directory/2

Description

`change_working_directory(NewPathName)`

Is equivalent with

```
change_working_directory(NewPathName, _)
```

`change_working_directory(NewPathName, OldPathName)`

Unifies **OldPathName** with an atom composed from the string returned by the operating system as the absolute pathname of the current working directory. If the unification fails, the predicate itself fails without changing anything. Otherwise instructs the operating system to change the current working directory according to the pathname constituting the **NewPathName** atom (either absolute or relative).

Template and modes

`change_working_directory(+atom)`

`change_working_directory(+atom, -atom)`

Examples

```
change_working_directory('../test/data', Previous),
    current_working_directory(Current),
    format('previous wd: ~a, current wd: ~a~n',[Previous, Current]).
```

Writes out something like:

```
previous wd: /usr/home/ks/csp/run, current wd: /usr/home/ks/csp/test/data
```

Errors

`type_error(atom, NewPathName)`
NewPathName is not an atom.

`type_error(variable, OldPathName)`
OldPathName is not an uninstantiated variable at the time of the call.

`system_error`
The operating system indicated an error (insufficient access rights, incorrect pathname or non-existing path element).
ErrInfo-Other contains the error number obtained from the operating system.

system/0

system/1

system/2

Description

`system`

`system(CommandString)`

`system(CommandString, Status)`

system/0 starts a new interactive default shell process; the control is returned to Prolog (with success) upon termination of the shell process.

system/1 and **system/2** both pass **CommandString** to a new default shell process for execution.

system/1 succeeds if the return status value is 0 after the completion of the shell, otherwise fails.

system/2 always succeeds and unifies the **Status** argument with a term describing the outcome of the command execution. The status term has one of the following forms:

```
exitstatus(N)
termsig(S)
syserr(E)
```

If the shell process terminated normally, **exitstatus(N)** is returned, where **N** is the exit code that the child process passed to *_exit()* or returned from its *main* function. Otherwise if the child process was terminated due to the receipt of a signal then **termsig(S)** is returned where **S** is the numeric code of the signal. Finally, if the execution of CommandString could not begin then **syserr(E)** is returned where **E** is the error number indicating the reason of refusal by the system.

Template and modes

```
system
system(+atom)
system(+atom, -term)
```

Examples

```
system('ps -e | fgrep csp > running.txt').
```

Uses system services (SunOs in this example) to prepare a file containing information lines about currently running operating system processes such that the substring **'csp'** appears within the line. Succeeds if at least one such line is found, otherwise fails.

Errors

```
type_error(atom, CommandString)
```

CommandString is not an atom.

```
type_error(variable, Status)
```

Status is not an uninstantiated variable at the time of the call.

tempname/1

tempname /2

tempname /3

Description

```
tempname(FilePathName)
```

Is equivalent with

```
tempname(NewPathName, '')
```

```
tempname(FilePathName, DirPath)
```

Is equivalent with

```
tempname(NewPathName, DirPath, '')
```

```
tempname(FilePathName, DirPath, Prefix)
```

These predicates provide an interface to the *tmpnam()* and *tempnam()* functions of the operating system. They generate file names that can be safely used for temporary files. Each call with the same arguments generates a different name (but after a system-wide limit is reached the names begin to cycle). The file names are temporary only in the sense that that they reside in a directory intended for temporary use; and they are likely to be unique. It is the user's responsibility to remove any file created with one of these names when its use is ended.

tempname/1 always generates a file name using the system-wide path-prefix set by the administrator at system generation time.

tempname/2 and **tempname/3** allow the user to control the choice of a directory. The argument **DirPath** specifies the name of the directory in which the file is to be created. If the **DirPath** string is not a name for an

appropriate directory, the operating system uses its own default instead (governed by the TMPDIR environment variable in the user's environment, the system-wide default set by the administrator, and using **/tmp** as a last resort).

Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the **Prefix** argument for this. (At most five characters from the front are used as the first few characters of the temporary-file name.)

The predicates should never fail in normal circumstances. If, however, the operating system for some reason cannot find an appropriate directory, the call will fail.

Template and modes

```
tempname(-atom)
tempname(-atom, +atom)
tempname(-atom, +atom, +atom)
```

Examples

```
tempname(FN), write(FN), nl.
```

Writes out something like:

```
'/tmp/baaa000M3'
```

```
tempname(FN, '../test/data', 'TmP'), write(FN), nl.
```

Writes out something like:

```
'../test/data/TmPJAAa000LV'
```

Errors

```
type_error(variable, FilePathName)
```

FilePathName is not an uninstantiated variable at the time of the call.

```
instantiation_error
```

DirPath or **Prefix** is an unbound variable.

```
type_error(atom, NewPathName)
```

DirPath is neither an atom nor a variable.

```
type_error(atom, Prefix)
```

Prefix is neither an atom nor a variable.

31. Predicates for the CLP extension

The Constraint Logic Programming (CLP) extension model in CSP-II allows for installing several CLP solvers in a CS-Prolog runtime program. Each solver can use its own set of interface predicates. There is, however, a so called normal CLP interface predicate set, supported and partially implemented by the base (core) system. It is recommended that each solver implement these predicates unless they are totally unfit for the purposes of that particular solver (see chapter 39 for details).

This chapter includes the description of the second and third group of the system provided CLP interface predicates. The predicates of the first group (extended type testing) are merged into chapter 13. Those predicates the name of which begins with '**clp_**' constitute the third group, the set of 'normal' CLP interface predicates.

If a normal interface predicate call is issued by the program and the currently selected solver does not support that call then a specific error is raised.

The set of normal interface predicates, defined originally for the ML Linear Solver, reflects the needs of the linear programming constraint handling model. This chapter describes the generic behavior of the predicates at the level as the system support is involved. The actually supported subset and the solver-specific details should be described for each solver separately.

In general (depending on the features of the installed solvers), each CS-Prolog process can have active instances of several different solvers, and each solver can have running instances in several CS-Prolog processes. The models (satisfiable set of constraints), maintained by different instances of a particular solver, are independent of each other.

All normal interface predicates except **clp_debug/1** are backtrackable. Specifically for this group of predicates this means that if the successful evaluation of a predicate causes any change in the model maintained by the selected instance, then after backtracking over the call the status reverts to the status that was in effect before the call. (Solver-specific foreign interface predicates must also be backtrackable.)

Errors

The following errors can be raised for any of the normal interface predicates. They will not be repeated in the individual descriptions. Note that all errors described in this chapter are raised by the CLP organizer component of the runtime system (core); the individual solvers can raise additional errors.

`system_error`
The runtime system does not include the CLP extension. **Other_info** contains the atom **`clp_extension_is_excluded_from_this_runtime`.**

`domain_error(installed_clp_solvers, Sid)`
There is no solver configured in the system that would match **Sid**.

`clp_system_error`
No solver is configured in the CSP runtime system. **Other_info** contains the atom **`no_solver_is_installed`**.

`clp_system_error`
The called normal interface predicate is not supported by the currently selected solver. **Other_info** contains the atom **`solver_does_not_support_this_predicate`**.

`clp_system_error`
The selected solver cannot be run in the current CS-Prolog process either because of some conflict with an instance of an other solver already running in the process, or because the selected solver does not support multithreading and already has a running instance in an other process (depending on features specified in the configuration table). **Other_info** is a two-item list where the first item is the atom **`cannot_create_solver_instance`**; the second item is one of the atoms **`second_instance_of_non_multithreaded_solver`**, **`thread_is_monopolized_by_another_solver`**, or **`this_solver_must_run_in_a_thread_alone`**, to describe the particular conflict type.

`clp_system_error`
Programming error related to the internal interface between the CLP organizer and the selected solver. **Other_info** contains a list of the form **`[internal_error,ErrCode]`**, where **ErrCode** is an integer code specifying the particular error detected (these errors should not occur in connection with a debugged solver; they are designed to help the development process). The meaning of the currently used codes are summarized in the following table:

ErrCode	Cause
1710	**sign_on** called before instance initialization
1720	Missing **sign_off** (2nd **sign_on** call with same **sid**)
1730	Missing **sign_off** from an other solver (2nd **sign_on** call with different **sid**)
1740	**sign_off** without **sign_on** (no solver is current)
1750	**sign_off** without **sign_on** (other solver is current)
1800	Invalid solver ID encountered
1820	2nd arg of **remove_last_restr_vars** is too big (greater than the number of 'live' c-vars)
1850	**put_data_on_trail** request from unidentified solver
1855	Invalid trail note request (backtrack inhibited)
1885	1st arg of **restr_var_ind** callback is not a constrained variable
1890	1st arg of **mk_restr_var** is not an unbound variable
1891	3rd arg of **mk_restr_var** (problem variable index) is not less than CLP_MAX_SOLVER_VARS (2048 at present)
1892	The number of currently 'live' problem variables for the solver exceeds CLP_MAX_PROBLEM_VARS (1024 at present)

NOTES:

- If the solver's implementation contains errors in specific components, then CLP-related error exceptions can be raised during backtracking, too. In this case, the **`Goal`** component of the additional information associated with the exception indicates the predicate call that triggered the backtracking by failing; most probably, this call is not directly related to the error condition.
- Some internal errors also might be reported for calls attempting unification where a constrained variable is involved.

query_clp_config/4

Description

`query_clp_config(Sid, Name, Coop, IsMultiThreading)`

For each solver configured in the runtime program this resatisfiable predicate unifies the arguments with attributes of the solver in the following way:

Sid is unified with the numeric identifier assigned to the solver (the 0-based index of the entry in the configuration table).

Name is unified with the mnemonic name specified in the configuration table.

Coop is unified with an atom that characterizes the 'cooperativeness' of the solver, from the set monopolistic (must run alone in a thread), xenophobic (does not recognize 'alien' constrained variables – the system must pre-filter all expressions), tolerant (recognizes and rejects alien constrained variables), cooperative (can handle alien variables).

IsMultiThreading is unified with either the atom **true** (if the solver is prepared to run in several instances) or the atom **false** (if it can run only in one CS-Prolog process).

The attributes of each installed solver are enumerated on backtrack.

Template and modes

```
query_clp_config(?term, ?term, ?term, ?term)
```

Examples

```
query_clp_config(Id, Name, _, _),
      format('Installed solver: Id=~d, Mnemonic name=''~a'''~n', [Id, Name]),
      fail.
```

Writes the identifier integer value and the mnemonic name of each installed solver to the current output, on separate lines, and finally fails.

```
query_clp_config(0, _, _, true).
```

Succeeds if there is at least one solver configured in the runtime program and the default solver (with **`Sid`** = **`0`**) supports multithreading, otherwise fails.

Errors

None

select_clp_solver/[0,1]

Description

```
select_clp_solver
```

is equivalent with

```
      select_clp_solver(0)
```

```
select_clp_solver(Solver)
```

This predicate is entirely implemented by the core system. It can be used by the program for selecting the solver for which the ensuing normal interface predicate calls (in 'forward' execution, within the same CS-Prolog process) should be dispatched. The **Solver** argument can be an integer for the SolverId (SolverIds are consecutive integers from 0 to N – 1, where ***N*** is the number of configured solvers), or an atom containing the symbolic name of the desired solver, specified in the configuration table entry. SolverId-s are essentially the index values for selecting the configuration table entry. The first entry (indexed by 0) is for the 'default' solver.

At the start of each process, the default solver (if there is one) is selected implicitly for the process.

The predicate is backtrackable.

Template and modes

```
select_clp_solver
select_clp_solver(+atom_or_integer)
```

Examples

```
select_clp_solver(ml_solver).
```

Succeeds if there is a solver configured with mnemonic name **`ml_solver`** in the system; otherwise an exception is raised. After successful execution the ensuing normal interface predicate calls (in forward execution) will be directed to this solver.

```
select_clp_solver(1).
```

Selects the second installed solver (with SolverId = 1) as target for the ensuing normal CLP interface calls. Succeeds if there are at least two solvers configured in the system; otherwise an exception is raised.

Errors

```
instantiation_error
```

The **Sid** argument is an unbound variable.

clp_constraint/1

Description

`clp_constraint(ConstrList)`

This is the central element of the CLP interface, used for defining constraints for the currently selected solver.

ConstrList is a list of structures (specific for the solver involved), each structure representing one constraint. For example, the ML linear solver accepts structures where the main functor is one of **(=:=) /2**, **(=<) /2**, or **(>=) /2**, and the arguments of each structure are *CLP linear expressions*.

The solver analyzes the constraints contained in **ConstrList**, decides whether they are syntactically correct and if so, whether adding them to the current set of constraints yields a consistent state.

If any error is detected, then an appropriate error is raised.

Otherwise, if the new set of constraints is inconsistent with the current status, then the call fails.

If no error is detected and the new constraints are accepted, then the call succeeds. Unbound variables encountered in the new constraints become constrained variables (corresponding to new problem variables).

The predicate is backtrackable. After backtracking over the call the status of the model maintained by the selected solver instance reverts to the status that was in effect before the call.

Template and modes

`clp_constraint(+list)`

Errors

`instantiation_error`
The **ConstrList** argument is an unbound variable.
`type_error(list,ConstrList)`
ConstrList is not a list.
`domain_error(valid_formula_for_clp_solver,ConstrList)`
There is an 'alien' constrained variable inside **ConstrList** and the currently selected solver is described as xenophobic. **Other_info** is a list of two atoms: **`[SolverName, constrained_variable_for_other_solver]`**, where **SolverName** is the mnemonic name assigned to the selected solver.

clp_type/[2,3]

Description

`clp_type(Expr, Type)`

is equivalent with

```
clp_type(Expr, Type, _)
```

`clp_type(Expr, Type, Extra)`

Expr should be an expression recognized by the solver, not containing unbound variables. The solver qualifies the expression (based on the current status of the constrained variables occurring in it), and returns an atom representing the result of the classification. The returned value is unified with **Type**. If any useful additional information can be appended, it will be unified with **Extra** as a list.

For example, the ML solver accepts CLP linear expressions, and returns one of the following atoms for **Type**: **free**, **lobnd**, **upbnd**, **bounded**, **fix**, **number**, where **number** means that **Expr** is a fully evaluable arithmetic expression (with constant value), the other categories correspond to value ranges like the ones described for the possible states of constrained variables.

Template and modes

`clp_type(+term, ?atom)`

`clp_type(+term, ?atom, ?List)`

Errors

`instantiation_error`
The **Expr** argument is an unbound variable.

`type_error(clp_evaluable_expression,Expr)`
Expr is a symbol or a list.

`domain_error(valid_formula_for_clp_solver,Expr)`
There is an 'alien' constrained variable inside **Expr** and the currently selected solver is described as xenophobic.
Other_info contains the atom **`constrained_variable_for_other_solver`**.

clp_max/[2,4]

Description

`clp_max(Expr, Value)`

is equivalent with

```
clp_max(Expr, Value, [], _)
```

`clp_max(Expr, Value, Query, Answer)`

Expr should be an expression recognized by the solver, not containing unbound variables. The solver attempts to calculate the maximal value that the expression can assume subject to the current set of constraints. If the maximum does not exist (the expression has no upper bound) then the call fails, otherwise **Value** is unified with the calculated maximal value. The **Query** argument can contain solver-specific query items (one item, or a list of items) about the solution of the current set of constraints corresponding to the maximum found. **Answer** is unified with the item, or with a list of items, that supply the answers to the query item(s) contained in **Query** (see also **clp_value/2**).

Template and modes

`clp_max(+term, ?number)`

`clp_max(+term, ?number, +term, -term)`

Errors

`instantiation_error`
The **Expr** argument is an unbound variable.

`type_error(clp_evaluable_expression,Expr)`
Expr is a symbol or a list.

`domain_error(valid_formula_for_clp_solver,Expr)`
There is an 'alien' constrained variable inside **Expr** and the currently selected solver is described as xenophobic.
Other_info contains the atom **`constrained_variable_for_other_solver`**.

clp_min/[2,4]

Description

These predicates are essentially the same as **clp_max/[2,4]** above, considering the equivalence

min { f(x) } == - max { -f(x) }

Template and modes

`clp_min(+term, ?number)`

`clp_min(+term, ?number, +term, -term)`

Errors

`instantiation_error`
The **Expr** argument is an unbound variable.

`type_error(clp_evaluable_expression,Expr)`
Expr is a symbol or a list.
`domain_error(valid_formula_for_clp_solver,Expr)`
There is an 'alien' constrained variable inside **Expr** and the currently selected solver is described as xenophobic.
Other_info contains the atom **`constrained_variable_for_other_solver`**.

clp_value/2

Description

`clp_value(Query, Answer)`

Query is an expression recognized by the solver, not containing unbound variables, or a list of such expressions. If **Query** is a structure (one expression), then the solver evaluates the expression, substituting values for the constrained variables in the expression from the feasible solution maintained as part of the current state, and unifies **Answer** with the result of this evaluation.

If **Query** is a list of expressions then the solver builds a corresponding list of results evaluating each expression as in the previous case, and unifies **Answer** with this list.

Template and modes

`clp_value(+struct_or_list, ?number_or_list)`

Errors

`instantiation_error`
The **Query** argument is an unbound variable.
`type_error(clp_evaluable_expression,Flags)`
Query is a symbol.
`domain_error(valid_formula_for_clp_solver,Query)`
There is an 'alien' constrained variable inside **Query** and the currently selected solver is described as xenophobic.
Other_info contains the atom **`constrained_variable_for_other_solver`**.

clp_debug_mode/1

Description

`clp_debug_mode(Flags)`

Passes the value of **Flags** to the selected solver. The intent of this predicate is to give the user program some control over any debugging facility provided by the specific solver.

This predicate is not considered as 'significant' (does not activate the solver instance in the CS-Prolog process if it is not activated yet; the **Flags** value is saved for later instead), and is not backtrackable.

Template and modes

`clp_debug_mode(+non_negative_integer)`

Errors

`instantiation_error`
The **Flags** argument is an unbound variable.
`type_error(integer,Flags)`
Flags is neither an integer nor an unbound variable.
`domain_error(not_less_than_zero,Flags)`
Flags is a negative integer.

PART IV

CS-Prolog development system

32. Files and directories

The CS-Prolog development system incorporates four programs and several other data files. The **compiler** compiles a CS-Prolog module into a binary module format. The **linker** connects several binary modules into a binary program file. This binary program file can be executed by the **runtime system**. The **programming environment** incorporates the previous three programs, making comfortable the process of programming. If the modules are compiled and linked using a special option, the CS-Prolog program can be debugged with an interactive trace facility.

The names of the CS-Prolog components are:

cspcomp - the compiler
csplink - the linker
csprolog - the runtime system
cspenv - the programming environment

The components of the CS-Prolog development system use several data files. In order to be able to find them, these files must reside in a directory that is known by the system. This directory is called **csphome** and the system is informed about the name of **csphome** through an environment variable **CSPHOME**. Therefore, before the first use of a CS-Prolog component, this environment variable has to be set (in an operating system dependent way) to the directory name of **csphome** directory. If this environment variable is undefined, the system searches the files in the current working directory.

The data files used by the CS-Prolog system are:

`xxpro.blt`
: A text file with the list of built-in predicates. It is required by the compiler.

`standard.mdf`
: A binary module containing the standard module. It is required by the linker.

`csptrace.mdf`
: A binary module that contains the debugger module. It is required by the linker (only if debug is wanted).

`prolint.pdf`
: A binary program called within the environment.

The user can extend CS-Prolog with his or her own built-in predicates written in C. The system files of the C interface are:

`csprolog.h`
: The C header file.

`csprolog.EXT`
: The CS-Prolog library (**EXT** is the (system dependent) extension for libraries).

`extname`
: Executable program for printing out the converted identifier if a CS-Prolog predicate name is not a valid C identifier (see section 38.1).

The output files of the compiler and the linker are system independent. That means that any binary module or binary program file can be ported to another machine and used there.

33. Compiler

The compiler compiles a CS-Prolog source module to a binary module format. The compiler has two passes. In the first one is performed the syntactical and semantic check. The second pass generates the binary module code.

The compiler can be executed with a following command issued in the host operating system:

```
cspcomp source [opt_list]
```

source is the file name of the source CS-Prolog module. The default extension for Prolog files is **pro**, if **source** has no extension this default is used. A successful compilation generates a binary output file (input file of a subsequent linkage). The extension of this generated file is **mdf**. The file name (without extension) of the generated file is that of **`source`**, unless a specific option is given. **opt_list** is a sequence of valid compiler options; this can be omitted if there are no options needed.

If the compiler detects an error in the source code, an error message is written to the standard output. There are three types of errors. If a fatal I/O error occurs (e.g. there is no file with the given name), the compilation is aborted. If a syntax error or a semantic error is found, no code is generated but the whole source file is processed. The errors are listed in Appendix A. Error messages have the following format.

For syntactic errors:

```
source(line_number): error number : error_text
```

For semantic errors:

```
error number : error_text
```

A syntax error means that a term in the source file violates the rules of CS-Prolog term or module syntax (see chapter 1). Semantic error is signaled if there are defined contradicting definitions for a predicate e.g. a functor is declared as dynamic and imported simultaneously. If a clause is very large or too complex, it can cause a code generation error.

The compiler performs a very useful check for undefined predicates. In CS-Prolog, the undefined calls are assumed as calls of dynamic predicates that will be created during runtime. In many cases, however, such undefined calls are consequences of some clerical error. Therefore the compiler displays a warning for these calls, but this warning does not have effect on the success of the compilation. (If the warnings are not needed, they can be suppressed.)

The valid options of the compiler are:

`-ptrace`
: Generate code that enables the CS-Prolog debugger to set breakpoints in this module. (The trace facility can handle modules compiled without this option if the program is linked with **`ptrace`** option).

`-fo filename`
: Generate output file in **`filename`**. If **`filename`** terminates with a path separator character (**`/`** or **`\`**) then it must be a directory name, and the output will be placed in this directory using the name of the source and the default extension (**mdf**). Otherwise **`filename`** is interpreted as a file name, if it does not have an extension, the default extension is suffixed.

`-nopp`
: Do not use the preprocessor.

`-nocode`
: Do not generate code file. Only syntax and semantic check is performed.

`-noch[eck_singleton]`
: Do not check singleton variables. Without this option specified the compiler generates a warning message if a non-void variable appears only once in a clause. Variables beginning with **_** (underscore) are not subject to this check. The option name can be abbreviated as indicated.

`-now[arn]`
: Do not write warning messages. The option name can be abbreviated as indicated.

`-P`

Perform only preprocessing, the preprocessed source is written out to a file with extension **i**. (A ***source*.i** file is created.) Neither syntax checking, nor semantic checking, nor code generation is performed.

Several options affect only the preprocessor. These are:

`-D`**`mac=value`**

Defines a macro with name **`mac`** and body **`value`**. If the **`=value`** is omitted the macro is defined as the text **`0`** (zero).

`-I`**`dir_name`**

The directory **`dir_name`** is prefixed to the list of directories where include files are searched.

`-M`

The pathnames of the source file and of all the included files (direct or indirect inclusion) are written to the standard output. This feature helps, e.g., in finding dependencies when a make script is prepared.

The compiler needs some data files as it is described in chapter 31.

34. Linker

The linker links several compiled binary CS-Prolog modules into a binary program file. This file can be executed by the runtime system. The linkage is done in two passes. In the first pass the export/import interfaces are checked, the appropriate predicates are connected, and the symbols of modules are merged into a common table. In the second pass the binary program file is generated. If there are foreign directive(s) in some modules, an additional file (C source) is generated (see chapter 38. describing the C interface).

The linker can be executed with a following command issued in the host operating system:

```
csplink [options] mod1 mod2 ... modN [,prog_name]
```

mod$_1$, **mod$_2$**, ..., **mod$_N$** are usually the file names of module files to be linked. They have a default extension (**mdf**); they should be files generated by the compiler. **prog_name** is the file name for the program binary file. If **prog_name** is omitted, then the name of the first module (**mod$_1$**) is used (without extension) by default. The output file also has default extension (**pdf**), so normally a ***prog_name*.pdf** (or ***mod$_1$*.pdf**) file is created.

If any of the **mod$_1$**, **mod$_2$**, ... , **mod$_N$** arguments begins with the distinguishing character **@** then it specifies an indirect input file called *response file* containing additional arguments. The **@** is not part of the file name.

A response file can contain arguments of the same form as allowed for direct command line arguments except that further indirection is not handled, and layout characters including newline can be used freely. The effect of a command line containing response file arguments is the same as if these arguments were replaced by the content of the corresponding response files except for line size limitation (included newline does not terminate the command).

The fatal I/O errors abort the linkage. If there are other linkage errors found, they are written out to the standard output and no program code is produced. All input modules must have been compiled with a version of the CS-Prolog compiler compatible with the linker version and the standard module version used. The linkage errors are listed in Appendix B.

Options can appear anywhere in the argument list. The valid options of the linker are:

`-ptrace`
: Generate code that enables the CS-Prolog debugger to trace the program. (It means including a special module for tracing in the generated program.)

`-nocr`
: Suppresses the display of the copyright notice by the linker.

35. Runtime system

The runtime system executes the CS-Prolog binary program files created by the linker. The main goal of the program (**main_goal/[0,1]**) is invoked after loading the program file. The termination of a CS-Prolog program can be successful, unsuccessful or caused by an unhandled exception. If all processes of the program terminate with success, it means the successful termination of the whole program and no message appears. If some of the processes fail, the system writes out a warning message. If an unhandled exception occurs, the system writes out the error term and breaks the execution.

The runtime system can be executed with a following command issued in the host operating system:

```
csprolog [opt_list]
```

opt_list is a sequence of valid runtime options; this can be omitted if there are no options. The CS-Prolog program to be executed is given with an option (**cspprog**).

The memory used by the Prolog system is determined in the beginning of the execution (can be modified with an option, see later). This limitation can cause a runtime error **resource(memory)** if all memory is used up. Therefore, it is essential to specify the amount of memory needed correctly, since it cannot be changed during execution. The memory available for the system is divided into two parts. First part is used by the main stacks of the processes. The second part is used for the symbols, floats, dynamic clauses, values and channels and messages created during the execution. The deeper is the recursion of the program the more memory is needed for the main stacks, the more data (listed above) are created the more memory is needed for them. So the division of the memory can be essential too, as this partitioning is decided in the beginning of the execution and cannot be changed later.

The runtime system accepts several options. Each runtime option has a name. Some of them also take a value; others have no value, only their presence or absence conveys the intended meaning (switches). The options can be specified in two different ways, either on the command line or in operating system environment variables.

If specified on the command line, a value taking option can be given as:

```
-Option_name=Option_value
```

or

```
-Option_name Option_value
```

If using the environment, the variable has to be named **Option_name** (upper case) and set to value **Option_value**. The command line option overrides the value set in then environment. On the command line the **Option_name** is case-insensitive, but the corresponding environment variable name has to be written always in upper case. Otherwise the form is similar except for the **'-'** character introducing the command line options (distinguishing them from the regular arguments passed to the program).

The options which need no value (switches) can be given on the command line simply with their name:

```
-Option_name
```

To set these options in the environment, a special variable named **CSPOPT** is used. Its value should consist of switch names separated by colons.

The valid options that have a value are:

```
cspprog=Binfile
cspmem=Number
gcstat=Number
```

The valid switches are:

```
ptrace
mintables
medtables
ver
h[elp]
```

With the **cspprog** option it is defined the binary program file to be executed by the runtime system. The extension **pdf** can be omitted. So to execute a program called **hello.pdf** the following options can be used:

```
-cspprog=hello
-cspprog hello
```

Alternatively, the user can set the variable **CSPPROG** (in a machine dependent way) to the value **hello**. The default value for this option is **csppr.pdf**.

The **cspmem** option specifies the amount of memory to be used by CS-Prolog. The default (if the option is not present) is machine dependent, on processors that support virtual memory management, it usually is 2048 Kbytes. The **Number** value of this option has to be an integer; it is interpreted in Kbytes.

The **ptrace** option requests Prolog trace, i.e., the runtime system will activate the interactive trace facility. However, the trace can be accessed only if the program had been linked with the same option. The trace can debug modules compiled without the **ptrace** option, but in this case the user cannot set breakpoints. (For the description of the interactive trace facility see chapter 37.)

The **gcstat** option activates a simple profiling tool that prints out a short summary of resource usage before and after garbage collection is performed. The **Number** value of this option regulates the frequency of this printing. **Number** = *K* means that each *K*-th garbage collecting action will be reported. **Number** = 0 disables this facility (the same as omitting the option).

The **medtables** and **mintables** options specify the ratio of division of the available memory between the main stacks and the other data. If neither of them is present, the ratio is the following: 70% for the main stacks and 30% for other data. Specifying the **medtables** option changes this ratio to 50%-50%, and with **mintables** the ratio will be 30%-70%.

The **ver** (**ver**sion info) option causes the printing of version identification data on the standard error stream.

The special **h**[**elp**] option causes the printing of a brief summary of the command line syntax and the available options. All other command line arguments are ignored, and the execution is terminated immediately after the printing. Any valid abbreviation of the **help** keyword is accepted.

If the main goal of the CS-Prolog program has arity 1, the system calls this **main_goal/1** predicate with a list argument composed of the regular (non-option) command line arguments. All command line arguments that begin with a minus sign character, and cannot be interpreted as a negative number, are treated as options — information for the runtime system itself. Arguments representing integer numbers are converted to CS-Prolog integers; all other arguments will be passed as atoms.

Example: the command

```
csprolog -cspprog test1 -cspmem 4000 foo bar -2 0 -ptrace
```

calls the CSP-II runtime system, which will first try to load the binary file **test1.pdf**, then synthesizes and performs the following call (in interactive debugging mode):

```
main_goal([foo, bar, -2, 0]).
```

The system will use 4 megabytes of memory if the operating system allows.

36. Programming environment

The programming environment combines the components of the CS-Prolog system. It permits to define CS-Prolog programs, to generate code for them, to run and to debug them. It does not incorporate a text editor; it is assumed that the source modules are written and modified with an editor existing on the host machine. On operating systems with multitasking possibilities, the editing can be performed by another task, independently from the CS-Prolog environment.

The environment creates a so-called make file for every CS-Prolog program. Executing the **make** utility of the operating system, supplying this make file, will invoke the CS-Prolog compiler and the linker. The user does not need to know anything about make files since the environment applies them automatically. Not all operating systems provide a make utility. The environment is able to compile and link the program without invoking a make utility if an appropriate option is set.

In the CS-Prolog environment there is always a so-called actual CS-Prolog program. If a command in the environment is issued without parameter, it will be interpreted as a command for the actual program. A command with a program name argument or an explicit **define** command changes the actual program.

The environment allows setting the command line options for the component programs. By default, the system generates code with trace information. It is also possible to change the name of the Prolog runtime system used, if the user created an extended CS-Prolog runtime system with the external C interface.

The environment system can be executed with a following command issued in the host operating system:

```
cspenv [prog_name]
```

prog_name is an optional argument. If it is present, the system executes a

```
make prog_name
```

command in the beginning, so sets **prog_name** to be the actual program, compiles and links the modules if the program is not up to date. See the next chapter for the description of the **make** environment command.

Some options can be changed in the environment (see the **set** command). When the user quits the environment and there were changes in some options, the system creates a so-called init file with extension **ini**, in the current working directory, and stores the changed options in this file. If such an init file is found in the directory the environment is invoked from, the previous options are automatically restored.

36.1 Environment commands

The commands can have optional arguments. In this description, the optional arguments are enclosed in brackets ([]). Every command can be abbreviated. The part of the command name which is after the abbreviation is enclosed in curly brackets ({}).

d{efine} [prog_name]

Defines the actual CS-Prolog program as the target of the development. **prog_name** is a file name for the program binary file **without extension**. If **prog_name** is omitted it is asked interactively

The command prompts first for the program name if it is not supplied as an argument. If there is a make file found for this program, the environment assumes that the program is already defined. (To define an existing program with modified module names see **redefine** command.)

If there is no make file for the program, then the module names constituting the program are asked. These names are file names, not the Prolog module names inside the files. To finish the definition after the last module name, an empty line should be given. If the program consists of a single module with the same name (and **pro** extension) as **prog_name**, then when the first module name is asked, an empty line should be entered.

When the definition has terminated the environment creates a make file for the program (if it is not yet present). The name of the make file is **prog_name.mak**.

If an existing make file is used the system checks if the program is up to date, or the modules have to be compiled and linked because they have been modified since the last compilation.

re{define} [prog_name]

Performs the same task as the **define** command, with the difference that module name(s) are requested even if there is a make file present for **prog_name**. Therefore, an existing program can be redefined to contain new module(s).

m{ake} [prog_name]

Generates code for the program **prog_name** or the actual program. If the **prog_name** argument is present then an implicit **define** command is executed. Code generation means the compilation of (modified) modules and the linkage if the compilation(s) were successful.

r{un} [prog_name]

Executes the actual program or the program **prog_name**. If the **prog_name** argument is present, an implicit **make** (and eventually **define**) command is executed. The execution is without trace.

tr{ace} [prog_name]

Executes with the trace facility the actual program or the program **prog_name**. If the **prog_name** argument is present, an implicit **make** (and eventually **define**) command is executed. For the description of the interactive trace see chapter 37.

p{rolog}

Executes a special CS-Prolog program. This program simply reads a term - a Prolog goal - and executes this goal. Displays the success or the failure of the execution, and, in the successful case, writes out the instantiation(s) of the variable(s) of the goal. It is possible to ask for another solution, that is, force the system to backtrack into the goal and display if it can succeed once more.

To give a Prolog goal at the **?-** prompt, simply type in the goal terminated with a period (end token). When a solution is displayed (the variable bindings are written out), the system waits for input. Type in a semi-colon (**;**) character to have a new solution, pressing **Enter** finishes the execution of the goal.

If an exception occurs in the goal, the programs writes out the error term, breaks the execution and returns to the **?-** prompt.

There are some special goals in this program:

`trace.`
: The subsequent goals will be traced with the interactive trace.

`notrace.`
: The subsequent goals will not be traced with the interactive trace. (This is the default.)

`load_file(FileName).`
: Reads in a text file **FileName** containing a (part of) a CS-Prolog module. All directives are ignored except the operator directives. No modules are created; all clauses are added into the module where the interactive trace is defined. Since there is already a **main_goal** in the program, an eventual **main_goal/[0,1]** predicate is renamed to **main__goal/[0,1]** (with double underscore in it). The added predicates will be dynamic and not static.

`quit.`
: Terminates the program.

h{elp} [command]

Gives general help information or help on a specific command.

q{uit}

Terminates the environment.

set c{ompopt} OPT
set l{inkopt} OPT
set r{unopt} OPT

Sets command line options for the compiler, linker or the runtime system. The default is for compiler and for the linker:

```
-ptrace
```

For the runtime system, if it is executed by **trace** command, the same option is given. For non-tracing execution there is no default command line option.

set p{rolog} prolog_file

Sets the name of the runtime system to be called by the **run** or **trace** command. By default it is **csprolog**, but if the user creates a new runtime system with the external C interface, its name can be set with this command.

set m{ake} [make_name]

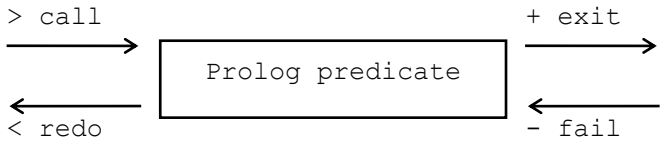
Sets the name of the make utility to be used by the **make** command. If **make_name** is omitted, no make utility will be used; the compilation and linkage will be invoked from the environment directly. On operating systems where there is provided a make utility the default value is **make**, on operating systems where there is no make utility the default is the direct invocation.

st{atus}

Prints out the name of the actual program, the module names, and the actual values of the options.

37. Debugging

The CS-Prolog trace facility uses the so-called **box model** to represent the flow of control through a Prolog predicate. This model visualizes the Prolog predicates as boxes with two entry ports and two exit ports, through which the control may pass.



In this figure, the box represents a Prolog predicate (with some clauses). The arrows indicate the direction of the flow of control through the ports. The characters **>**, **+**, **<** and **-** are the symbols of the four ports named **call**, **exit**, **redo** and **fail**.

The **call** port represents the initial invocation of the predicate. The **exit** port represents the successful satisfaction of the predicate. The **redo** port represents the backtracking into the predicate after a failure in order to find an alternative solution. The **fail** port represents the failure of the predicate, when the backtracking continues.

When tracing a predicate in the CS-Prolog interactive trace the call is displayed in all four ports. On the **entry** port the execution stops, the trace waits for a trace command. A predicate can defined to be a break point. It makes possible to execute the program without trace until a call is reached and then to continue with tracing.

The following trace commands can be given. The commands can be abbreviated, the part of the command that is not needed is enclosed in curly brackets ({}).

`{trace}`
: Single step execution. Trace will stop at the entry port of the predicate called next. (The abbreviation of this command is a simple **Enter** key.)

`s{kip}`
: Skip the trace of the current call. Trace will stop at the entry port of the predicate called next, after the termination (successful or failed) of the current call.

`g{o}`
: Go without trace. Trace will stop at the entry port of the next break point.

`b{reak} Functor`
: Set break point at the predicate with functor **`Functor`**. **`Functor`** can contain a module prefix, if it doesn't, the predicate is searched in the current module. If the arity part of the **`Functor`** is omitted (it is an atom), then the predicate with this name which is found first, is set to be a break point. The **Functor** has to be in a single line.

  Important note: on multiprocessor environments setting a break point means setting it on the actual processor. Therefore, a process running on an other processor will not stop on this predicate. Since dynamic predicates are local to a process, the break points on dynamic predicates are local to a process as well.

`c{lear} Functor`
: Remove break point. **`Functor`** has to be given in the same way as in the previous command. The predicate with functor **`Functor`** will not be a break point any longer.

`q(uit)`
: Abort the execution.

`p{rolog}`
: Enter Prolog mode. Offers the same facility as the environment command **prolog**, described in the previous section. So CS-Prolog goals can be executed. To set the module where the goals are interpreted use a module prefix or the

  ```
  defmod(Mod_name).
  ```

  special goal. After executing this call, all subsequent goals will be invoked in the module **`Mod_name`**.

The trace indicates in its output the processor and the process where the goal is running. This information has format **`Pr/Ps`** displayed after the traced call, where **`Pr`** is the sequence number of the processor and **`Ps`** is the sequence number of the process. The processor part is omitted if the process runs on the main processor, and everything is omitted in case of the main process.

38. The C interface

The CS-PROLOG system provides the means for writing user-defined built-in predicates in C language (*foreign* predicates). A special C function set is supplied for accessing arguments, unifying, etc. A C function can be called from a CS-Prolog program if the user performs the following:

Writes and compiles his or her own C source(s) using the header file **csprolog.h**, generating some C object file(s).

Adds **foreign** directive(s) to the Prolog source(s) for each foreign predicate referenced, and compiles the Prolog source(s).

Links the CS-Prolog program (the linker automatically will generate a C source file with name **cspfor.c**).

Compiles the **cspfor.c** source generated by the linker, creating another C object file.

Links the C object files with the CS-Prolog library to create a new version of the runtime system, containing the new predicate(s).

In this section, a basic knowledge of the C programming language is assumed.

An essential restriction for the implementation of such built-in predicates is that the C representation of CS-Prolog terms must not be saved in global variables by the program for later use. This representation may contain pointers that become invalid during garbage collection (the garbage collection procedure is not aware of these stored terms). In special cases (backtrackable and non-deterministic predicates), the C interface gives the possibility of storing and retrieving CS-Prolog objects from internal stacks.

38.1 The prototype of the C function

The name of the C function implementing a Prolog built-in predicate is derived from the functor of the predicate, connecting the name and the arity with a **_c_** character sequence. Therefore, if a functor **Name/Arity** is declared as a foreign predicate (see section 1.2.1), the name of the corresponding C predicate should be:

Name1_c_**Arity**

This C function must not have arguments, and has to return an integer. For a foreign predicate **plus/3** the prototype of the C function to be written is:

```
int plus_c_3(void);
```

(In some non-ANSI C implementations, the **void** keyword has to be omitted.)

If the predicate name **Name** is not longer than 25 characters and constitutes a valid C identifier then **Name1** is the same as **Name**. Otherwise, **Name1** is an artificially constructed identifier. The utility program **extname** can be used to obtain this identifier. This program reads a line from the standard input and prints out the converted identifier on the standard output. If the CS-Prolog name is a valid C identifier, this identifier is printed. The names of external C predicates can also inspected in the file **cspfor.c** generated by the linker (the names are listed as initializer values for the elements of a C array).

Note that, at present, the linker automatically generates the **cspfor.c** file each time when a program containing at least one **foreign** directive is linked. This file lists only those predicates that are actually mentioned in the program, so the customized runtime system produced using this file will know about these predicates only. An alternative is to prepare a dummy CSP source module that lists all foreign predicates implemented by the user as a package, to prepare the runtime program using the **cspfor.c** file produced by compiling and linking this dummy source, and to ignore the **cspfor.c** files generated for the 'real' programs. The runtime in this case will accept foreign calls to a superset of those needed for individual programs.

The arguments of the Prolog predicate can be accessed inside the body of the C functions through interface functions described in the next section.

A special type named CspRetC is defined (as *integer*) for the values returned by the functions. Two special return values of the C function indicate the success or the failure of the predicate. Any other return value raises the corresponding exception.

38.2 Basic C definitions

The C source file implementing user-defined builtin predicates normally should include the following header file shipped with the distributed system

```
#include "csprolog.h"
```

This file defines the basic Prolog structures, interface functions, and some data items.

The basic data type for the internal data representation of CS-Prolog is

```
xxp_cell
```

Every Prolog term in C has the type **xxp_cell**. There is a special xxp_cell denoting the empty list, it is defined by a macro. Its name is

```
XXP_NIL
```

The actual Prolog term represented by the current content of an **xxp_cell** as seen in the external C module can be classified into one of the following categories: unbound variable, integer number, float number, atom, list, or structure (non-list compound term). The symbolic values (macro names) for these categories are:

```
XXP_T_EMPTYREF
XXP_T_INT
XXP_T_FLOAT
XXP_T_SYMB
XXP_T_LIST
XXP_T_STRUCT
```

The version of CS-Prolog extended for CLP support (Constraint Logic Programming) defines an additional term type, the so-called constrained variable, which belongs to the category denoted by the symbol

```
XXP_T_RESTR
```

Prolog integers can be retrieved in C as long integers, float numbers as double-s.

A special data type,

```
CspRetC
```

is defined for the values returned by the built in predicates (and some other functions).

The **CspRetC** values for indicating the success or failure of the built-in predicate are:

```
XXP_E_SUCCEED
XXP_E_FAIL
```

The C function has to return **XXP_E_SUCCEED** if it succeeds, and **XXP_E_FAIL** to indicate failure. Any other value is interpreted as an error signal that causes the raising of the corresponding exception.

There is another type definition

```
CspFloatType
```

for the floating-point value representation used by the system (at present it is a synonym for the type *double*).

The header file contains prototypes for the C functions provided by the CS-Prolog runtime system. If the macro **__STDC__** is defined (possibly by the C compiler) the prototypes are ANSI C compatible.

38.3 The C interface function set

38.3.1 Functions accessing Prolog terms

```
xxp_cell xxp_arg(int arg_no);
```

Returns the **arg_no**-th argument of the built-in predicate. The arguments are numbered, as in Prolog, from 1.

```
int xxp_type(xxp_cell cell);
unsigned xxp_raw_type(xxp_cell cell);
```

Returns the type of **cell**:

```
XXP_T_EMPTYREF   unbound variable
XXP_T_RESTR      constrained variable (only with CLP extension)
XXP_T_INT        integer
XXP_T_FLOAT      floating point number
XXP_T_SYMB       atom (symbol)
XXP_T_LIST       list
XXP_T_STRUCT     compound term (not list)
```

The main difference between these two functions is that for the empty list (**XXP_NIL**) they return different values: **xxp_type** returns **XXP_T_LIST** while **xxp_raw_type** returns **XXP_T_SYMB**. The type of the returned value is also different (but the returned values are always non-negative for **xxp_type**, too).

The following C functions have an xxp_cell argument and return some information. All work only on a special type of cell (e.g. cell representing an integer or an atom). However, since no check is provided, if called with an inappropriate argument, the function will return false information. Therefore, it is very important to check the type of the cell before calling one of these functions.

```
long xxp_int_val(xxp_cell cell);
```

Returns the integer value of **cell** representing a Prolog integer. Works properly only for cells of type **XXP_T_INT**. (It is a macro, not a function.)

```
CspFloatType xxp_float_val(xxp_cell cell);
```

Returns the floating-point value of **cell**, representing a Prolog float number. Works properly only for cells of type **XXP_T_FLOAT**. For other types, an exception is raised internally.

```
CspFloatType xxp_checked_float_val(xxp_cell cell, unsigned argpos, CspRetC *rcp);
```

This is the generalized variant of the function xxp_float_val above; enables the caller to handle an eventual error directly. The second argument, **argpos**, specifies the argument position to be indicated in the error term in case of error. The behavior of the function depends on the value of the return code pointer **rcp**. If **rcp** is NULL then this function acts like xxp_float_val (except for the more specific argument position indication for error). Otherwise, when **rcp** is different from NULL, then the function either returns the floating-point value of the Prolog float number represented by of **cell**, and puts XXP_E_SUCCEED into the variable pointed at by rcp, or, when the type of cell is improper or some other error occurred, returns 1.0 (as a least harmful default), and places the code of the prepared exception into ***rcp**.

```
char * xxp_symb_chars(xxp_cell cell);
```

Returns a pointer to the characters of **cell** representing a Prolog atom. Works properly only for cells of type **XXP_T_SYMB**. The pointer returned points into the internal Prolog data array. So the program should use the pointer only for accessing the characters, never should change the data.

```
xxp_cell xxp_list_head(xxp_cell cell);
```

Returns the head of **cell** representing a Prolog list. Works properly only for cells of type **XXP_T_LIST**.

```
xxp_cell xxp_list_tail(xxp_cell cell);
```

Returns the tail of **cell** representing a Prolog list. Works properly only for cells of type **XXP_T_LIST**.

```
xxp_cell xxp_struct_functor(xxp_cell cell);
```

If type of **cell** is **XXP_T_STRUCT** then returns the name (principal functor) of the structure represented by **cell**. The returned cell represents an atom (symbol), but also contains information about the arity of the functor. It can be compared with a functor created with **xxp_mk_functor** function. If the type is **XXP_T_LIST** then returns the *dot functor* (**'.')/2** which corresponds to the symbol **'.'**. For any other type, the function returns the **cell** argument itself.

```
unsigned xxp_struct_arity(xxp_cell cell);
```

If type of **cell** is **XXP_T_STRUCT** then returns the arity of the Prolog compound term represented by **cell**. If the type is **XXP_T_LIST** then returns **2**, corresponding to the *dot functor* **('.')/2**. For any other type, the function returns **0**.

```
xxp_cell xxp_struct_arg(int argno, xxp_cell cell);
```

If type of **cell** is **XXP_T_STRUCT** then returns the **argno**-th argument of the Prolog compound term represented by **cell**. The arguments are numbered beginning with 1, up to the arity of the structure. If the type is **XXP_T_LIST** then for **argno** = 1 returns the head of the list represented by **cell**; for **argno** = 2 returns the tail of the list. For any other type, and for **argno** values outside the allowed range, the result is undefined and the call may cause an exception.

38.3.2 Functions for creating Prolog terms

```
xxp_cell xxp_mk_emptyref();
```

Creates and returns a new, unbound variable.

```
xxp_cell xxp_mk_int(long n);
```

Returns a cell representing the integer number **n** as a Prolog integer (this is a macro, not a function). If the value of **n** is outside of the range of Prolog integers, then it is 'folded' into the range.

```
xxp_cell xxp_checked _mk_int(long n, unsigned argpos, CspRetC *rcp);
```

This is the enhanced version of the **xxp_mk_int** function above; enables the caller to detect and handle the condition when the value of **n** falls outside of the range of Prolog integers. When **n** is within the allowed range then the result is the same as from **xxp_mk_int**. Otherwise the behavior of the function depends on the value of the return code pointer **rcp**. If **rcp** is NULL then integer overflow error is raised internally by the function, and the control will not return to the caller. If **rcp** is different from NULL then the code of the prepared error signal is placed into ***rcp** and the function returns a cell representing the integer value 1 (as the least harmful default). The error term signaled or prepared will indicate **argpos** as the argument position.

```
xxp_cell xxp _mk_float(CspFloatType f);
```

Returns a cell representing the float number **f**.

```
xxp_cell xxp_mk_symb(const char * chars_adr);
```

Returns a cell representing a Prolog atom, which contains the characters pointed at by **chars_adr** (a null-terminated string). The length of the string, including the terminating null character, must not exceed the limit defined as MAX_ATOM_LENGTH in the header file. The **const** is omitted in non-ANSI C compilation.

```
xxp_cell xxp_mk_functor(const char * chars_adr, int argno);
```

Returns a cell representing a Prolog functor that contains the characters pointed at by **chars_adr** and has arity **argno**. (See also **xxp_mk_symb** above.)

```
xxp_cell xxp_mk_list(xxp_cell head, xxp_cell tail);
```

Returns a cell representing a Prolog list with head **head** and tail **tail**.

```
xxp_cell xxp_mk_struct(int argno, xxp_cell name, xxp_cell *args);
```

Returns a cell representing a compound term, which has arity **argno**, name **name**, and whose arguments are in array **args**.

38.3.3 Functions for unification

```
CspRetC xxp_unify(xxp_cell term1, xxp_cell term2);
```

term1 is unified with **term2**. It returns **XXP_E_SUCCEED** if the unification was successful and **XXP_E_FAIL** otherwise. If the unification fails then the variable bindings are not undone; that means that if this function call fails then the built in predicate must return **XXP_E_FAIL** as well.

```
CspRetC xxp_unify_with_occurs_check(xxp_cell term1, xxp_cell term2);
```

Performs unification of **term1** and **term2** with occurs check. It means that if **term1** and **term2** are not unifiable or **term1** and **term2** are unifiable and no cyclic term is created during unification then this function has the same effect as **xxp_unify** above. If during the unification a cyclic term would be created, then the function returns **XXP_E_FAIL** (the variable bindings made before cycle detection are not undone).

```
CspRetC xxp_safe_unify(xxp_cell term1, xxp_cell term2);
```

term1 is unified with **term2**. It returns **XXP_E_SUCCEED** if the unification was successful and **XXP_E_FAIL** otherwise. If the unification fails then the variable bindings are undone. That means that if this function call fails then the built in predicate can continue and eventually succeed.

38.3.4 Functions for non-deterministic predicates

The user has the possibility to implement foreign built-in predicates that can succeed more than once. These functions create a so-called choice point, where the information needed by the backtracking mechanism of CS-Prolog is stored. After successful termination of a non-deterministic C function (and Prolog foreign predicate), a subsequent failure will call the same C function again. The choice point has to be created by the first invocation of the function and has to be destroyed explicitly by the last call (decision of the called function). Of course, a Prolog cut (!) can destroy the choice point as well.

The non-deterministic predicates usually need some data that can be accessed and modified. These data cannot be generally stored in static C variables, because there can be more than one choice point present for the same predicate. Any such information needed for the next call of the function can be stored only in **xxp_cell**-s that are put in the choice points. These cells are considered as virtual arguments.

For every non-deterministic built-in predicate, a C structure has to be statically declared. The type of this structure is

```
extb_choice_point
```

This structure is used by the runtime system, and the user must not change the value of its fields.

The following C functions serve for creating a non-deterministic foreign predicate.

```
int xxp_first_call();
```

Returns **1** (TRUE) if the call of the foreign predicate is entered in 'forward' execution (first time). Otherwise, when the call is entered during backtracking as for a new alternative, the function returns **0** (FALSE).

```
void xxp_create_choice_point(int n, extb_choice_point *chp);
```

Creates a choice point accommodating **n** argument cells. **n** has to be greater than, or equal to, the arity of the predicate. Let's denote this arity by **ari**. The arguments of the predicate are stored in the first **ari** cells; the rest **n-ari** cells can be used for storing data to be retained between the successive calls. **chp** is a pointer to a static structure. Never use a pointer to an automatic variable!

```
void xxp_set_choice_point_arg(int argno, xxp_cell cell);
```

Sets the **argno**-th cell in the choice point to **cell**. Do not use this function for **argno** less or equal then the arity of the predicate (it would change an argument originated from Prolog). **argno** has to be less or equal then the size of the choice point set in call of **xxp_create_choice_point**.

```
void xxp_destroy_choice_point();
```

Removes the choice point. It has to be called in the last call of the predicate (when the called function decides that there are no more alternative solutions).

38.3.5 Functions for backtrackable predicates

The user has the possibility to implement foreign built-in predicates that undo something when the system backtracks on them. These functions can push some data - named trail block - on an internal stack, if information is needed for undoing. When backtracking, the trail block is passed as an argument to a user-defined C function that is responsible for the undo operation. This function is called undo function. Every backtrackable predicate implementation must register itself in the system; by doing so, it gets a unique trail

identifier. Every call of such a predicate makes then a so-called trail note supplying its identifier and the current data - the trail block. (More exactly: trail notes can be set up only using a trail identifier obtained by registering a trail note format and undo function; several predicates can share the same identifier if otherwise appropriate).

The trail block, which is an array of **xxp_cell**s, can contain two types of data: CS-Prolog terms, or something else not interpreted by the CS-Prolog engine, e.g. a flag-set or a pointer, explicitly cast to the type **xxp_cell**. If CS-Prolog terms are stored in a trail-block, the system has to be informed about this fact, because the garbage collection has to deal with all living CS-Prolog terms. So when a backtrackable predicate registers itself, it must give the information, which element of its trail blocks are CS-Prolog terms.

The prototype of the registering function:

```
typedef int (*UndoRoutine)(xxp_cell * tr_block);
int xxp_register_user_trail_block(int block_size,
                        long mask, UndoRoutine undo_func);
```

The function returns an integer serving for identification of the trail notes of this predicate. **block_size** is the number of cells stored in a trail block. The returned integer is always greater than zero. **mask** is an integer, in which one bit is set to 1 for each cell in the trail block that is to contain a CS-Prolog term. (The least significant bit corresponds to the 0th element of the block.) E.g., if the size of the block is 3, and all elements are CS-Prolog terms the mask has to be 7, if only the second element is a CS-Prolog term, then the mask should be 4. **undo_func** is a function that will be called every time, when backtrack occurs on this particular predicate. The argument of the **undo_func** will be the trail block. (The value returned by **undo_func** is ignored at present.)

```
void xxp_make_trail_note(int note, xxp_cell *tr_block);
```

Makes a trail note - pushes the trail block on the internal stack. **note** is the identifier returned by a previous **xxp_register_user_trail_block** call. **tr_block** is the actual trail block. It has to be of the exact size specified in the registration (no check is done), and can contain CS-Prolog terms only on positions given in the registered mask (no check can be done either).

38.3.6 Memory handling

On platforms with limited memory (e.g. transputers), the CS-Prolog system allocates all memory available when initializing the runtime system. So the standard C memory handling functions **malloc** and **free** cannot be used. On machines with virtual memory management these standard C functions will work, but for compatibility purposes it is recommended to call the interface functions that allocate (and free) memory maintained by the CS-Prolog runtime system.

```
char * xxp_allocate_user_block(unsigned size);
```

Allocates a piece of memory of size **size**. It is a replacement of the standard **malloc** function. Returns the pointer to the newly allocated memory.

```
void xxp_free_user_block(char *ptr);
```

Frees memory allocated previously with **xxp_allocate_user_block**. It is a replacement of the standard **free** function.

38.3.7 Raising exceptions

If the foreign predicate detects an error, it can be signaled in several ways. The necessary information can be prepared either directly by the function, in which case an arbitrary error_term can be composed, or one of the standard Prolog exceptions formats, listed in the section 4.3, can be prepared using the corresponding error composing function.

In the first case a C function has to be called which will not return (executes a **setjmp** C library routine inside).

```
void xxp_do_signal_prepared(xxp_cell err_term, xxp_cell info_term);
```

Raises an exception with error term **err_term** and additional error info term **info_term**. **err_term** and **info_term** are arbitrary Prolog terms created by the user. The function does not return. If the error will be handled by **protected/3** predicate, the **info_term** should be of the same format as described in section 4.3, a list with the erroneous call as first element.

```
xxp_cell xxp_construct_builtin_call(void);
```

Returns the CS-Prolog term representation of the built-in call from which it is invoked. This term can be used to construct the error info term passed to a **xxp_do_signal_prepared** call.

In the second case, the standard exception information can be prepared by using the corresponding C function (from the list below), which returns an error code (of type CspRetC). As side effect of this call, the exception information is stored in a static data structure in the system. The prepared information will be used for raising the exception, if that happens 'shortly' after the call (normally before, or at, returning from the predicate). The following C functions prepare error information and return standard error codes:

```
CspRetC xxp_instantiation_error(unsigned argno,xxp_cell other);
CspRetC xxp_type_error(unsigned argno, unsigned valid_type, xxp_cell culprit,
            xxp_cell other);
CspRetC xxp_domain_error(unsigned argno, unsigned valid_domain, xxp_cell culprit,
            xxp_cell other);
CspRetC xxp_existence_error(unsigned argno, unsigned object_type,
            xxp_cell culprit, xxp_cell other);
CspRetC xxp_permission_error(unsigned operation, unsigned permission_type,
            xxp_cell culprit, xxp_cell other);
CspRetC xxp_representation_error(unsigned argno, unsigned flag,
            xxp_cell other);
CspRetC xxp_syntax_error(xxp_cell atom, xxp_cell other);
CspRetC xxp_resource_error(unsigned resource, xxp_cell other);


CspRetC xxp_system_error(xxp_cell other);
CspRetC xxp_interrupt(unsigned interrupt_name, xxp_cell interrupt_data);
```

For these functions, argument **argno** shows which argument caused the error, argument **culprit** is a Prolog term representing the culprit in the error term, and argument **other** is a Prolog term representing the last argument of the error term. The values of arguments **valid_type**, **valid_domain**, **object_type**, **operation**, **permission_type** **flag**, **resource**, and **interrupt_name** have to be one of the constants defined in the **csprolog.h** header file for the given error kind.

Note that if **other** is a term the type of which is different from XXP_T_LIST, then the system will convert it into a one-item list, or, for **xxp_clp_sy_error**, if it is an integer, then into a two-item list of the form **[internal_error,Value]**.

The preferred way of raising the prepared exception is to return the error code obtained, as the outcome of the predicate call. In this case, the exception will be raised by the CS-Prolog engine immediately after the call.

As an alternative, the following function can be used:

```
void xxp_signal(CspRetC signal_code);
```

This function is similar to **xxp_do_signal_prepared**, but uses the information prepared inside the system instead of passing the terms as argument. The function works properly only if the stored information is still valid and corresponds to **signal_code**. This condition can be ensured if **signal_code** is obtained in the proper way (from one of the error composing functions, or as the outcome of another system-provided function; no other error has been prepared since its formation; and it has not been stored across different invocations of the function implementing the built in predicate). Direct usage of **xxp_signal** instead of returning **signal_code** as the outcome of the predicate might be necessary when the error is detected in a deeply embedded subroutine call and passing the code all the way upward is not organized, and there is no need to reset the internal state of the implementing module.

There are three other functions logically belonging to this group of exception handling. They can be used for coping with errors returned from other (usually system-provided) functions. The simple handling of such errors is to return from the predicate immediately, passing the error code received as the outcome of the predicate, or raising the corresponding signal directly. There are, however, circumstances when the predicate itself can handle the error or wants to change the exception indicated. The functions supporting this activity are the following:

```
xxp_cell xxp_get_error_term(CspRetC signal_code);
```

This function returns a cell that would constitute the **error_term** if **signal_code** were signaled at that moment. **signal_code** is supposed to be a standard error code obtained recently. The term returned is built from the error information stored in the system by the latest error-composing function call. If the stored information does not correspond to **signal_code** then XXP_NIL is returned. This can happen if an intervening error-composing call overlaid the original data, if task switching or garbage collection has been performed since the preparation (neither should occur during one invocation of an external function), or if the error has been reset (see below). The returned term can be inspected by the caller for learning the details of the error.

```
xxp_cell xxp_get_other_for_error(CspRetC signal_code);
```

This function is similar to **xxp_get_error_term** above; the difference is that a cell representing **other_info** is returned (instead of **error_term**).

```
void xxp_reset_error_flags(void);
```

This function might be used if the predicate handles locally the standard error prepared most recently. It is not strictly necessary to call the function even in this case, because normally the next error will overlay the prepared error data. The most useful effect of the call is that an ensuing **xxp_get_error_term** call will return XXP_NIL. After resetting the error, the error code pertaining to the prepared signal must not be used for raising the signal.

38.3.8 Generating events and interrupts

It is possible to generate an event or an interrupt from a C function imitating the CS-Prolog predicates **generate_event/[1,2]** and **cause_interrupt/2**.

```
int xxp_generate_event(xxp_cell name, xxp_cell data);
```

> Generates an event named **name** and having event data **data**. These arguments are Prolog terms created by the user. The function will cause an exception when called with invalid arguments (see description of **generate_event/2**).

```
int xxp_cause_interrupt(xxp_cell proc, xxp_cell data);
```

> Generates an interrupt for the process named **proc**, with interrupt data **data**. The function will cause an exception when called with invalid arguments (see description of **cause_interrupt/2**).

38.3.9 Calling a Prolog predicate from C

There is an interface function for evaluating a CS-Prolog goal from C code.

```
int xxp_call_prolog_from_C(xxp_cell *goal)
```

> Calls the term pointed by **goal** as a meta-call. If this term does not contain a module prefix, the interpretation of the call is tried in the current module first, and, if it is not defined there, in the first module where an appropriate - public - Prolog predicate is defined. If the goal term contains a module prefix, only that module is used.
>
> The function returns XXP_E_SUCCEED if the Prolog call terminated successfully and returns XXP_E_FAIL if the call failed. If the goal term does not represent a legal Prolog call, the function returns a suitable error code. However, the exceptions and interrupts that occur during the execution of the Prolog call have to be handled with usual CS-Prolog tools (**protected/3**, **catch/3**) inside the execution. The protection levels, which were set previously, are not valid during the - inner - call.

The error code returned by the interface function when the goal term is not a legal Prolog call, can be passed as the return code of the C function implementing the foreign Prolog predicate, so that this exception will be raised at the embedding Prolog level. If the C program has to know what type of error occurred, it should use the

```
XXP_ERROR_KIND(ret)
```

macro for extracting a coded value, which is one of the integers **XXP_INSTANTIATION_ERROR**, **XXP_TYPE_ERROR** etc. defined in the header file. There is an internal static storage to store exception details, so a new exception overwrites the data of the previous one. So, do not keep error code results for long!

If the C function stores in **xxp_cell** variables some Prolog terms (atoms, structures, etc.) and then **xxp_call_prolog_from_C** is called, the cells might cease to represent valid Prolog terms after returning from this call because of an eventual garbage collection. So, never store Prolog terms in C variables across this function call! If a term has to be retained it should be passed as argument to the ***goal**, so that garbage collection would be aware of it. You can always use Prolog data items accessed through **xxp_arg** (with a valid argument number); furthermore after calling **xxp_call_prolog_from_C** the components of ***goal** term are valid up to the next such call.

The inner Prolog call is performed as part of the active process. All persistent changes (clauses or values modified) take effect normally. Unification of (sub)terms of arguments passed to the calling foreign predicate will have the usual effect as well after returning.

If the active process is suspended inside an inner (called from C) Prolog call, other processes can execute a new call Prolog from C. The order of returns from these inner calls is fixed, the **call_prolog_from_C** function called last has to return first. Therefore, it can occur that some processes are waiting for the termination of an inner Prolog call of another process. These waiting processes are reported to be in **buried** state.

The allowed number of arguments in **goal** is reduced by 1; it becomes actually 254.

38.4 Foreign predicate example

The program below implements five foreign predicates with the external C interface of CS-Prolog.

```
#include <string.h>
#include "csprolog.h"
/*
 *  Returns in the third argument the result of
 *  addition of the first two arguments
 */
int plus_c_3()
{
  long l;
  xxp_cell arg1 = xxp_arg(1);
  xxp_cell arg2 = xxp_arg(2);

  if (xxp_type(arg1) != XXP_T_INT)
    return(xxp_type_error(1,EXC_VTYPE_INTEGER,arg1,
                          XXP_NIL));
  if (xxp_type(arg2) != XXP_T_INT)
    return(xxp_type_error(2,EXC_VTYPE_INTEGER,arg2,
                          XXP_NIL));
  l = xxp_int_val(arg1) + xxp_int_val(arg2);
  return xxp_unify(xxp_arg(3),xxp_mk_int(l));
}

/*
 *  Duplicates the first argument and unifies
 *  the duplicate with the second argument
 */
#define MAX_SYM_LEN 1024
xxp_cell list_append();
#define MAX_ARG 16

int duplicate_c_2()
{
  xxp_cell out_cell;
  xxp_cell arg = xxp_arg(1);
  char dup_str[MAX_SYM_LEN], *str;
  xxp_cell dup_args[MAX_ARG];
  int i, arity;

  switch (xxp_type(arg)) {
    case XXP_T_INT :
      out_cell = xxp_mk_int(2 * xxp_int_val(arg)); break;
    case XXP_T_FLOAT :
      out_cell = xxp_mk_float(2 * xxp_float_val(arg));
      break;
    case XXP_T_SYMB :
```

```
      str = xxp_symb_chars(arg);
      if (strlen(str) > MAX_SYM_LEN/2 - 1)
       return(xxp_representation_error(1,
                  EXC_FLAG_MAX_ATOM,XXP_NIL));
      strcpy(dup_str,str);
      strcat(dup_str,str);
      out_cell = xxp_mk_symb(dup_str); break;
    case XXP_T_LIST :
      out_cell = list_append(arg,arg); break;
    case XXP_T_STRUCT :
      arity = xxp_struct_arity(arg);
      if (arity > MAX_ARG/2)
        return(xxp_representation_error(1,
                   EXC_FLAG_MAX_ARITY,XXP_NIL));
      for (i = 0; i < arity; i++)
        dup_args[arity+i] = dup_args[i] =
                              xxp_struct_arg(i+1,arg);
      out_cell = xxp_mk_struct(2*arity,
                    xxp_struct_functor(arg),dup_args);
      break;
    default :
      return XXP_E_FAIL;
  }
  return xxp_unify(xxp_arg(2),out_cell);
}

xxp_cell list_append(list1, list2)
  xxp_cell list1, list2;
{
  if (xxp_raw_type(list1) == XXP_T_LIST)
    return xxp_mk_list(xxp_list_head(list1),
       list_append(xxp_list_tail(list1),list2));
  else
    return list2;
}

/*
 *  Non deterministic predicate.
 *     range(N1,N2,X)
 *  returns in the third argument the numbers
 *     N1, N1+1, ... N2
 *  on backtracking.
 */
static extb_choice_point range_choice_point;

int range_c_3()
{
  xxp_cell arg1 = xxp_arg(1);
  xxp_cell arg2 = xxp_arg(2);
  long l2, l_out;

  if (xxp_first_call()) {
    if (xxp_type(arg1) != XXP_T_INT)
      return(xxp_type_error(1,EXC_VTYPE_INTEGER,arg1,
                            XXP_NIL));
    if (xxp_type(arg2) != XXP_T_INT)
      return(xxp_type_error(2,EXC_VTYPE_INTEGER,arg2,
                            XXP_NIL));
    l2 = xxp_int_val(arg2);
    l_out = xxp_int_val(arg1);
    if (l_out > l2)
      return XXP_E_FAIL;
    if (l_out == l2)
      return xxp_unify(xxp_arg(3),arg1);
    xxp_create_choice_point(4,&range_choice_point);
  }
  else {
    l2 = xxp_int_val(arg2);
    l_out = xxp_int_val(xxp_arg(4));
  }

 if (l_out == l2)
```

```
   xxp_destroy_choice_point();
 else
   xxp_set_choice_point_arg(4,xxp_mk_int(l_out + 1));

  return xxp_unify(xxp_arg(3),xxp_mk_int(l_out));
}

/*
 * Backtrackable example
 *
 * Simple global set_value_b and get_value_b
 * for atomic arguments on a single processor
 *     (The built-in set_value_b is local to a process)
 *
 * Stores the value in memory allocated from Prolog
 *
 * There is a fixed static array for values
 *     (could be made dynamic)
 *
 */
#define MAX_VAL 16

typedef struct {
  xxp_cell name;
  int      type;
  char *   value;
} v_type;

static v_type v_array[MAX_VAL];
static int    v_max = 0;

static int set_trail_note = 0;

#define SET_TR_NOTE_SIZE      3
#define SET_TR_NOTE_CELL_MASK 1L

#define NO_VALUE 0

static int undo_set_b(xxp_cell *trail_block);

int glob_set_b_c_2()
{
  xxp_cell name      = xxp_arg(1);
  xxp_cell new_value = xxp_arg(2);
  int tname = xxp_type(name);
  int tvalue = xxp_raw_type(new_value);
  xxp_cell tr_block[SET_TR_NOTE_SIZE];
  int i, int_val;
  unsigned size;
  char *symb_val, *val_ptr;
  double float_val;

  if (tname == XXP_T_EMPTYREF)
    return(xxp_instantiation_error(1,XXP_NIL));
  if (tvalue == XXP_T_EMPTYREF)
    return(xxp_instantiation_error(2,XXP_NIL));
  if (tname != XXP_T_INT)
    return(xxp_type_error(1,EXC_VTYPE_INTEGER,name,XXP_NIL));
  if (tvalue == XXP_T_LIST || tvalue == XXP_T_STRUCT)
    return(xxp_type_error(1,EXC_VTYPE_ATOMIC,name, XXP_NIL));

  if (set_trail_note == 0)
    set_trail_note = xxp_register_user_trail_block(
                       SET_TR_NOTE_SIZE,
                       SET_TR_NOTE_CELL_MASK,
                       undo_set_b);

  /* Find the name in the array */
  for(i = 0; i < v_max; i++)
    if (v_array[i].name == name)
      break;
  if (i == v_max) {       /* not found */
```

```
    if (v_max == MAX_VAL)
      return xxp_representation_error(1, EXC_FLAG_MAX_ATOM,XXP_NIL);
    v_max++;
    v_array[i].name = name;
    v_array[i].type = NO_VALUE;  /* for undo */
  }
  tr_block[0] = name;
  tr_block[1] = (xxp_cell)v_array[i].type;
  tr_block[2] = (xxp_cell)v_array[i].value;

  switch (tvalue) {
  case XXP_T_INT:
    int_val = xxp_int_val(new_value);
    val_ptr = (char *)&int_val;
    size = sizeof(long);
    break;
  case XXP_T_FLOAT:
    float_val = xxp_float_val(new_value);
    val_ptr = (char *)&float_val;
    size = sizeof(double);
    break;
  case XXP_T_SYMB:
    symb_val = xxp_symb_chars(new_value);
    val_ptr = symb_val;
    size = strlen(symb_val) + 1;
    break;
  }

  xxp_make_trail_note(set_trail_note,tr_block);

  v_array[i].type = tvalue;
  v_array[i].value = xxp_allocate_user_block(size);

  if (v_array[i].value == NULL)
    return xxp_resource_error(EXC_RESOU_MEMORY,XXP_NIL);
  memcpy(v_array[i].value,val_ptr,size);

  return XXP_E_SUCCEED;
}

int glob_get_b_c_2()
{
  xxp_cell name = xxp_arg(1);
  xxp_cell value;
  int i;

  int tname = xxp_type(name);

  if (tname == XXP_T_EMPTYREF)
    return(xxp_instantiation_error(1,XXP_NIL));
  if (tname != XXP_T_INT)
    return(xxp_type_error(1,EXC_VTYPE_INTEGER,name,XXP_NIL));

  /* Find the name in the array */
  for(i = 0; i < v_max; i++)
    if (v_array[i].name == name)
      break;
  if (i == v_max)   /* not found */
    return xxp_permission_error(EXC_OPER_ACCESS, EXC_PERMTYPE_VALUE,
                                name, XXP_NIL);

  switch (v_array[i].type) {
  case XXP_T_INT:
    value = xxp_mk_int((*(long *)v_array[i].value));
    break;
  case XXP_T_FLOAT:
    value = xxp_mk_float((*(double *)v_array[i].value));
    break;
  case XXP_T_SYMB:
    value = xxp_mk_symb(v_array[i].value);
    break;
  }
```

```
  return xxp_unify(xxp_arg(2),value);
}

static int undo_set_b(xxp_cell *trail_block)
{
  xxp_cell name = trail_block[0];
  int i;

  /* Find the name in the array */
  for(i = 0; i < v_max; i++)
    if (v_array[i].name == name)
      break;
  if (i == v_max)   /* not found */
    return XXP_E_FAIL;  /* impossible, always found */

  xxp_free_user_block(v_array[i].value);

  if (trail_block[1] == NO_VALUE) { /*undo first set*/
    int j;
    for (j = i + 1; j < v_max; j++)
      v_array[j - 1] = v_array[j];
    v_max--;
  }
  else {
    v_array[i].type = (int)trail_block[1];
    v_array[i].value = (char *)trail_block[2];
  }
  return(XXP_E_SUCCEED);
}
```

39. The Constraint Logic Programming (CLP) extension

The Constraint Logic Programming paradigm had been introduced by J.Jaffar, in 1987. The CS-Prolog runtime program can be configured to include one or several different CLP solvers (the technical maximum at present is four solvers).

There is a user-tailorable customization source module in the distribution kit for this purpose, in which the desired solvers and their important properties can be specified (**clp_cfg.c**). The solvers themselves should be supplied in the form of linkable object libraries. The process of creating a customized runtime is much the same as in the case of extending the runtime with user-defined (foreign) built-in predicates. There is an experimental linear solver, called the ML solver, included in the distribution kit. This solver is based on a linear programming algorithm; it handles linear inequalities and equations over real numbers.

For the sake of simplicity, when describing the interaction between different parts of the system, we shall speak of 'the solver' and 'the core', meaning the subsystem that is responsible for maintaining and repeatedly re-evaluating the set of constraining conditions, on one hand, and the component that keeps track of the active instances of the different solvers, dispatches requests originated by the Prolog program to the appropriate instance, and performs other system-related tasks, on the other hand.

A couple of CLP-related predicates are defined in the new CSP built-in predicate set. They can be divided into three groups. The first group is concerned with the term type system extension, and contains the following predicates (see chapter 13):

```
constrained_var/1          % c.f. var/1
strict_nonvar/1            % c.f. nonvar/1
strict_ground/1            % c.f. ground/1
```

The second group consists of the solver-independent predicates used for obtaining information about the installed solvers and selecting a particular solver (see chapter 31):

```
query_clp_config/4
select_clp_solver/[0,1]
```

The third group consists of the 'normal' interface predicates used directly in the work with the solvers (see chapter 31):

```
clp_constraint/1,
clp_type/[2,3]
clp_max/[2,4]
clp_min/[2,4]
clp_value/2,
clp_debug_mode/1,
```

The predicates in the third group require cooperation between the core system and the particular solver (currently selected). The idea of separately selecting the current solver is borrowed from the input/output system where the program can select the current input stream and current output stream with **set_input** and **set_output**, respectively, and the non-specific forms of other i/o predicates implicitly refer to the streams 'set' previously. The differences are that the 'clp' predicates have only non-specific forms - they cannot specify directly the solver -, that **select_clp_solver** is backtrackable, while **set_input** and **set_output** are not, and that the selection is process-specific, not global.

The Prolog interface for a solver may consist of some or all of the 'normal' predicates and the set also may be extended by defining specific predicates as 'foreign' ones. In fact, the solver can entirely ignore the normal predicates.

Each solver has a **SolverId** assigned to it by the configuration module. The solver which has the value 0 assigned as **SolverId** is called the default solver; it is selected initially and can be selected by calling **select_clp_solver/0** (without argument). The SolverId is the index of the corresponding entry in the configuration table; the default solver is described by the first entry.

The actual properties of the solvers are specified in the configuration table; in the most general case several CS-Prolog processes can have instances of more than one solver running (but only one instance of each particular solver), and each solver can be active in more than one CS-Prolog process (multithreading), and can 'live together' with instances of other solvers in one thread. (Note that here and in the sequel, 'thread' is used as a synonym for 'CS-Prolog process' for short, not for the operating system's thread concept.) Different

instances of the same solver should be unrelated in the sense that problem-specific data from one thread should not be manipulated by an instance living on another thread.

The supposed general behavior of a solver instance is as follows.

The instance is initialized when the first 'significant' predicate call is issued by the user program in the thread. All normal interface predicates are deemed significant except **clp_debug/1**. If the solver defines its own interface predicates, then it must notify the core system if the predicate call qualifies as significant. The instance starts with an empty 'model' (system of constraints). During forward execution, new constraints are incrementally added to the model. The solver evaluates the resulting constraint set, and, if it proves feasible, accepts the additions (the call succeeds), otherwise rejects them (fails). If the predicate, which passes the new constraint - typically **clp_constraint** - succeeds, all unbound variables occurring in the passed constraints become transformed into *constrained variables* (see later in this chapter). If the Prolog program later performs backtrack over a **clp_constraint** call, the solver must revert to the state that was in effect before that call.

Beside **clp_constraint** or its solver-specific counterpart, adding special constraints equivalent with explicit constraints of the form

<constrained_variable> =:= <value>

or

<constrained_variable> = <value>

can also be originated from Prolog unification implicitly (the actual form depends on which relation is understood by the solver).

It is important that the 'observable' state of the model after backtracking should be identical with the state before the call the effect of which had just been undone by the backtrack procedure. (Internal details exposed by **clp_debug** are not considered as part of the observable state).

The rest of the interface predicates usually serve only for querying; they should not change the actual state of the model.

The **clp_constraint/1** predicate takes as argument a list of structures (specific for the solver involved), each structure representing one constraint. For example the ML linear solver accepts structures where the main functor is one of **(=:=) /2**, **(=<) /2**, or **(>=) /2**, and the arguments of the structure are *CLP linear expressions*, , i.e. extended arithmetic expressions that may contain unbound variables and constrained variables, but only in such a way that after 'flattening' the expression and evaluating the variable-free subexpressions, the result is a (possibly multi-variable) polynomial, each addend of which has a summary degree of at most one.

A separate document describing the interface for developing and attaching a new CLP solver is available for external developers (subject to a license agreement).

39.1 New term type: *constrained variable*

The CLP extension introduces this new type into the system of term-types. ***Constrained variable*** is a rather peculiar type; in some respects it is like a variable, in others it resembles a non-ground structure, and in still other circumstances it can behave like a number.

It is important to tell in advance that this extension changes the meaning of several built in predicates relying on the closed set of types defined by the Standard; a couple of assumptions valid in standard Prolog do not hold any more. The change, however, affects only those programs that actively use the extension (work with one or more solvers).

Constrained variables as a type have no literals or other external representation in the source program; they appear when an unbound variable is first time mentioned in a constraint passed to a solver (if the call succeeds). They are, of course, associated with a corresponding internal variable of the solver. We'll call these internal variables 'CLP problem variables' or simply 'problem variables', to distinguish them from those that are not associated with Prolog objects (as e.g. slack variables in the case of a linear solver).

In the term ordering sequence, used in term comparison, 'constrained variable' type is inserted between 'variable' and 'floating point number' (see chapter 15).

Solvers can differ in the numeric types they handle. From the system's point of view the important factor is that whether the solver uses strict - 'Prolog' like - typing (i.e. integers are different from floats, even if numerically equal) or flexible - 'arithmetic' - typing (only arithmetic equality is considered). With strict typing, it is up to the solver to accept or reject any particular numeric type (integer or float).

The ML solver uses 'arithmetic' typing. The following discussion is based on the ML solver.

The range of admissible values for a particular variable at the current state of the model with respect to the range of the underlying numeric type can be one of the following:

- totally unrestricted (the variable is said to be FREE);
- limited from one side (UPBND or LOBND, meaning that it has an upper or a lower bound, respectively);
- confined to a closed interval (BOUNDED);
- fixed at a particular value (FIXED).

The Prolog unification rules are extended for constrained variables. This extension is based on a supposed generic handling of problem variables by the solvers. According to the generic model, a problem variable (associated with a CS-Prolog constrained variable) can be in different states as the model is incrementally enriched (or 'narrowed').

The unification rules for constrained variables are the following:

- An unbound variable can always be unified with a constrained variable. After unification it simply refers to the latter.
- If a constrained variable is being unified with an other constrained variable belonging to the same solver instance, or with a numeric value and the corresponding problem variable is not FIXED, then the solver is called via a special entry point, and the terms to be unified are passed as arguments to this call. The solver should treat the call as a special form of constraint being passed from the user program, and either reject it because of being inconsistent, or add it to the set of constraints, changing the state of the model (or just accept, if the new condition is entailed by the current set).
- If a constrained variable is being unified with a numeric value, and the current status of the corresponding problem variable is FIXED (possibly after accepting the new restriction in the preceding step), then

  1) If the solver uses 'arithmetic' typing then the success or failure of the unification depends on whether the numeric value is arithmetically equal (**=:=/2**) with the value at which the problem variable is fixed.

  2) If the solver uses 'Prolog' typing, then instead of arithmetic equality the Prolog unifiability (**=/2**) is used in deciding success or failure.

  In both cases, unification is handled by the core; the solver (in this step) is only queried about the status of the problem variable, and its current value if the status is FIXED. (The kind of 'typing' used by the solver is also queried once, in the preparatory phase).

- In all other cases, the unification attempt fails.

If the unification with a numeric value succeeds then the constrained variable becomes transparently bound to the numeric value like a normal unbound variable.

Note that the solver itself cannot bind a constrained variable to a numeric value; this can happen only when the Prolog program requests unification. The main reason of this is that, for a solver using arithmetic typing, when the problem variable becomes FIXED, it is still not decided what type of numeric value should the constrained variable assume.

39.2 Special behavior of constrained variables

The fact that constrained variables are associated with 'problem variables' maintained by the solver, and the state of the solver being synchronized with the Prolog evaluation stack (backtracking), implies that constrained

variables have no meaning outside of the normal Prolog environment of the Prolog process where they originally came to being.

For this reason, whenever a term is detached from this environment, constrained variables should lose their specific meaning, and behave like any other unbound variable. Such detachment occurs, among others, in input/output, in message passing, in Prolog database modification, and in 'global' value handling.

The 'output' predicates (**write**, **format** & Co.) produce a special form of variable token for a constrained variable. This token differs from the tokens representing normal unbound variables in that they contain a "CVAR" sub-string, and also a sub-string showing the SolverId of the owning solver if it is not the default solver.

For message passing, Prolog database updating (**asserta** & Co), and for non-backtrackable 'global' value setting, constrained variables in the term are replaced with new unbound variables.

In backtrackable global value setting (**set_value_b/2**) constrained variables are retained (these values are kept synchronized with the evaluation stack).

Built-in predicates requiring a numeric value in some argument do not accept a constrained variable at that position even if the status of the variable is FIXED and the predicate otherwise would accept both integer and floating point argument in that argument. This restriction does not concern those variables that have been unified with a numeric value, as explained above; after unification they represent the number, not the problem variable.

Furthermore, a built-in predicate requiring an unbound variable for some argument will reject a constrained variable at that argument position.

Appendix A - Error messages of the compiler

Fatal error caused by system files

```
Cannot open built-in file
Cannot open operator file
```

Fatal errors

```
Module name expected
Incorrect export list
Multiple source
Missing argument
No input file name
Cannot open input file
Out of memory
```

Syntax errors

```
100    token is too long
101    string is too long
102    string ends illegally
103    invalid character code
104    invalid character
105    invalid esc sequence
106    number expected
107    integer out of range
108    float number out of range
109    denormalized float number
110    incorrect token
111    left parenth expected
112    right parenth expected
113    right curly bracket expected
114    right user bracket expected
115    end of term expected
116    comma expected
117    comma or period expected
118    comma or right parenth expected
119    comma or list-end token expected
120    list begins incorrectly
121    list ends incorrectly
122    too many arguments
123    unexpected end-of-file
124    unexpected period
125    incorrect term
126    not parsible as a number

200    module begins incorrectly
201    wrong mode specification
202    invalid module name for import source ('[]' or self)
203    wrong clause head
204    wrong clause body
205    incorrect functor list
206    invalid functor arity
207    non name in functor spec
208    non-negative integer expected
209    name expected
210    full predicate indicator expected
211    inp out inout symb expected
212    unknown directive or invalid arity
213    directive is reserved for internal use
214    unsupported directive
215    directive is not supported (use preprocessor facility)
216    repeated module directive
217    incorrect priority
218    incorrect operator type
219    comma operator cannot be redefined
220    conflicting infix and postfix specification
221    module name is too long
222    invalid bracket component
223    conflicting bracket exists
```

224 bracket and operator conflict
225 curly bracket cannot be redefined
226 empty list atom cannot be bracket
227 unknown compile-time prolog flag
228 unknown value for prolog flag
229 End-of-file inside comment
230 clause behind the logical end-of-program

Semantic warnings

1 no public predicates
2 **predicate ind** is undefined (taken for non-existing dynamic)
3 **predicate ind** is undefined, but has been mentioned as discontiguous (taken for dynamic)
4 **predicate ind** is undefined, but has mode declaration
5 **predicate ind** - redundant mode declaration
6 **predicate ind** is being imported; earlier 'discontiguous' directive is ignored
7 **predicate ind** is built-in predicate; 'discontiguous' directive is ignored
8 **predicate ind** - repeated export
9 **predicate ind** - repeated import
10 **predicate ind** - repeated mode declaration
11 **predicate ind** (clause-**N**) - singleton variable **variable name**
12 **predicate ind** (clause-**N**) - first (meta) call is an uninstantiated variable
13 **predicate ind** (clause-**N**) - contains uninstantiated variable as meta-call

Semantic errors

30 **predicate ind** is exported, but not defined
31 **predicate ind** built-in predicate is imported from user module
32 **predicate ind** - incompatible mode declaration
33 **predicate ind** is built-in predicate; it cannot be redefined
34 **predicate ind** - the directive must preceede the 1st clause of the procedure
35 **predicate ind** - discontiguous directive is invalid for external procedure
36 **predicate ind** - non-contiguous clause for the procedure
37 **predicate ind** - redundant or contradictory declaration
38 **predicate ind** already has local definition
39 **predicate ind** - already imported
40 **predicate ind** is already declared external
41 **predicate ind** is already declared foreign
42 **predicate ind** - conflict with earlier import
43 **predicate ind** - contradiction with earlier mode declaration
44 **predicate ind** - contradiction with other mode declaration for imported procedure
45 **predicate ind** is a control construct
46 **predicate ind** is a control construct; it cannot be redefined
47 **predicate ind** - import from self without renaming
48 **predicate ind** - import from self of non-exported procedure
49 '[]' is not allowed as module name
50 invalid module prefix (atom or variable expected)
51 invalid call (atom, structure, or variable expected)

Fatal code generation errors

Dynamic clause is too complex (needs too many XAM registers)!
Static clause is too complex (needs too many XAM registers)!

Code generation errors

```
predicate predicate ind, clause-N; limit of calls in clause exceded
predicate predicate ind, clause-N; limit of calls in path exceded
predicate predicate ind, clause-N; limit of paths in clause exceded
```

Code generation warnings:

```
no public predicates
```

Appendix B - Error messages of the linker

Fatal linkage errors

Error: missing argument
Error: path name is too long
Error: too many module names
Error: Invalid indirect filename argument '@'
Error: response file is too large
Error: item in response file is too long
Error: invalid argument list (extraneous comma)
Error: invalid argument list (more than one output file)
Error: cannot open temporary file
Error: cannot open output file **file_name**
Error: cannot open module file **file_name**
Error: cannot open output file cspfor.c
Error: invalid argument list
Memory full
Temporary file(s) cannot be removed
Error: Inconsistent module in **file_name**, wrong XAM_opc version
Error: Inconsistent module in **file_name**, wrong builtin set version
Error: Inconsistent module in **file_name**, wrong optional module header size
Error: Inconsistent module in **file_name**, compiled with an incompatible old compiler version
Error: *Unknown error* in **file_name**, in optional module header (sh. not occ)

Linkage errors

Error: main goal is not defined
Error: duplicate main goal definition
Error: duplicate module name **module_name**
Error: unknown module name **module_name** in module file **file_name**
Error: **predicate_ind** is not public; imported in module **module_name**
Error: Closed import loop detected
module **module_name_1** imports **predicate_ind_1a** as **predicate_ind_1b**
module **module_name_2** imports **predicate_ind_2a** as **predicate_ind_2b**
(…)
Error: Mode declaration for **predicate_ind_1** in module **file_name_1** does not agree with the definition **predicate_ind_2** in module **file_name_2** (**mode_string_1** instead of **mode_string_2**)
Error: illegal option: **option**

Linkage warnings

Warning: Mode declaration is missing for **predicate_ind** in module **file_name** (**mode_string**)
Warning: Import of **predicate_ind** in module **module_name** is ambiguous.
Primary supplier module found: **module_name_1**
Additional supplier module: **module_name_2**
(…)

Appendix C - Implementation of real-time processes

Some built in predicates are implemented in CS-Prolog (in the 'standard' module, which is linked to every executable program). Normally this should not concern the programmer. There are, however, specific circumstances under which the programmer might be confronted with the actual implementation details of built in predicates.

In some cases during symbolic debugging it is possible that trace goes into the Prolog-level execution of a built in predicate. It usually does not cause any problem - the programmer can simply let the program **go**.

Another such situation is when an interrupt occurs, and the interrupted goal is printed out for inspection (either because no handler is installed for that particular interrupt, or because the handler itself prints out the goal).

Sometimes the programmer might not understand at what point exactly has the program been interrupted. The internal names appearing might give some hint in most cases.

There is, however, an important built in predicate which is totally hidden from the user, namely the main goal of a real-time process implementing its cyclic behavior. For those interested in implementation details the corresponding part of the 'standard' module source is included below.

```
call_realtime_goal(EVENTGOAL, INITGOAL) :-
  INITGOAL, !, '$init_realtime_process',
    call_event_loop(EVENTGOAL),
    '$terminate_event'.                  % (last event)

call_event_loop(EVENTGOAL) :-
  \+(call_event_goal(EVENTGOAL)), !;     % backtrack (clean)
  call_event_loop(EVENTGOAL).            % cycle

call_event_goal(EVENTGOAL) :-
  '$receive_event', EVENTGOAL,
    '$terminate_event', !.
```

Notes:

Predicates whose name begin with **$** are implemented in C.

'**$receive_event**' is the asynchronous counterpart of **receive/2** as far as communication is concerned.

Appendix D - Changes between versions

Changes for version 2.1:

New predicates added: **ground/1**, **calendar_time/1**, **time_zone/2**, **localtime_conversion/2**, **gmtime_conversion/2**, **localtime_info/4**, **gmtime_info/3**, **localtime_atom/3**, **current_environment/2**, **current_working_directory/1**, **change_working_directory/[1,2]**, **system/[1,2,3]**, **tempname[1,2,3]**.

Chapter 24 dealing with date and time related predicates had been completely rewritten.

Chapter 30 (Miscellaneous predicates) is new.

The description of the following predicates had been updated to reflect the networking extension: **receive/[2,3,4]**, **test_channel/2**, **channel_list/1**. The description of the channel state record had been updated accordingly.

Some clerical errors had been corrected and an attempt to improve the clarity of the text had been undertaken.

Changes for version 2.2:

Redundant backslash characters inside quoted strings cause syntax error instead of being ignored.

ANSI standard type-checking predicate **float/1** is added as a synonym for **real/1**.

Out-of-range numeric values cause syntax error when read as (part of) terms and at compilation.

Term syntax is changed (in accordance with the Standard) so that atoms that are operator names, left bracket names, or right bracket names cannot be immediate arguments of an operator in operator notation (as atoms). In such position, these special names must be surrounded by parentheses.

In directives that take predicate indicators as argument, sequence of predicate indicators is also accepted besides list of predicate indicators.

A new option for the **`float_range_checking_function`** prolog flag had been introduced: **`underflow_to_zero_after_rounding`**.

The following new directives had been added (as required by the Standard): **set_prolog_flag**, **discontiguous**.

Directives defined in the Standard but unsupported in CSP-II are also recognized.

`at_end_of_stream` and **`past_end_of_stream`** properties of source streams are set only when the end of the stream is actually reached (earlier the fetching of the **`end_of_stream`** atom as a term by read used to set the indicator).

Control constructs and built-in procedures cannot be redefined by the user program any more.

The first occurrence of a directive specifying some property of a user-defined procedure must precede the first clause of that procedure in the source text (**dynamic**, **meta_predicete**, **discontiguous**).

The consistency of explicit mode-declarations across modules is checked by the linker; missing mode declarations are indicated by warnings.

The linker warns about ambiguous import (when the importing module does not specify the exporting module, and more than one modules export the requested procedure). The reference is solved to the first matching item found.

The chapter describing **`cut`** has been removed from the manual; a new section dealing with control construct has been included instead into chapter 1 (Syntax).

Flexible predicate bindings are now process-specific. Cyclic binding is prevented. The arguments of the **bind/2** predicate must be visible at the place of the call (with the exception that a restricted flexible predicate can be bound to a local procedures in the module to which it is restricted even from outside).

Changes for version 2.3:

CLP extension is prepared, the new ‘constrained variable’ term type is defined for it. New built-in predicates: **constrained_var/1**, **strict_nonvar/1**, **strict_ground/1**, **query_clp_config/4**, **select_clp_solver/[0,1]**, **clp_constraint/1**, **clp_type/[2,3]**, **clp_value/2**, **clp_max/[2,4]**, **clp_min/[2,4]**, **clp_debug_mode/**1.

Chapters 31 and 39 are new.

Chapter 38 describing the C interface has been updated.

New command line options for calling the runtime system: **-ver**, **-h[elp]**

Warning about singleton variables is now the default for the compiler; the **-check_singleton** option is replaced by the new **-nocheck_singleton** option that can be used to suppress these warnings.

Appendix E - Known problems and errors

localtime_atom/3 under SunOs 4 shows the following anomalies:

When the date value to be converted falls within the range of timestamp values handled by the operating system (13 December 1901, 22:45:52 - 19 January 2038, 03:14:07, UTC), but there is no information about daylight saving for that date in the database of the OS, then the name of the zone in effect at the time of the execution is used (for format item **%Z**) instead of the primary zone name, but the correction is performed always according to the value defined for the primary zone.

If the date is very close (within the amount of time zone correction) to the limits of the range handled by the OS then in some cases either wrap-around occurs from one limit to the other, or truncation to the nearest limit.

Index of Built-in Predicates

General Index