ALS Prolog Reference

Applied Logic Systems, Inc.
PO Box 175
Cambridge, MA 02140 USA
Email: info@als.com
WWW: http://www.als.com

Syntax

29 Prolog Language Syntax

The description following is a slightly simplified version of the ISO standard description of Prolog language syntax, with the ALS Prolog extensions and modification included.

29.1 Processor character set

The Prolog ***processor character set*** is divided into several subgroups:

- alpha numberic characters
- graphic characters
- solo characters
- layout characters
- meta characterrs

char ::=	graphic char | alpha numeric char | solo char | layout characters
graphic char :: =	"#" | "$" | "&" | "*" | "+" | "-" | "." | "/" | ":" | "<" | "=" | ">" | "?" | "@" | "^" | "~"

A graphic character denotes itself in a quoted character.

Alphanumeric characters

alpha numeric char::=	alpha char | decimal digit char
alpha char ::=	underscore char | letter char
letter char ::=	capital letter char || small letter char
small letter char ::=	"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
capital letter char ::=	"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" |

	"M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
decimal digit char ::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
octal digit char::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
hexadecimal digit char::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f"
underscore char ::=	"_"

An alphanumeric character denotes itself in a quoted character. Two atoms and/or variables that are adjacent must be separated by one or more layout character or a comment.

Solo characters

solo char ::=	cut char | open char | close char | comma char | semi-colon char | open list char | close list char | open curly char | close curly char | head tail separator char | end line comment char
cut char ::=	"!"
open char ::=	"("
close char ::=	")"
comma char ::=	","
semicolon char ::=	";"
open list char ::=	"["
close list char ::=	"]"
open curly char ::=	"{"

close curly char ::= "}"

head tail separator char ::= "|"

end line comment char ::= "%”

Layout characters

layout char ::= space char | new line char

space char ::= " "

new line char ::= ***implementation dependent***

A space char denotes itself in a quoted character. A new line char is not allowed in a quoted character. No ***unquoted*** layout character which occurs between tokens is ever itself a token or a part of any token.

Meta characters

meta char ::= backslash char | single quote char | double quote char | back quote char

backslash char ::= "\"

single quote char ::= "’"

double quote char ::= """

back quote char ::= "‘"

A ***meta character*** modifies the meaning of the following characters, for example

1. A backslash character starts an escape sequence in a quoted token, in a char code list token, and in a character code constant. But in a graphic token, it behaves like a graphic char.
2. A single quote char is used to indicate the start and end of a quoted token.
3. A double quote char is used to indicate the start and end of a char code list

token.

4. A back quote char is used to indicate the start and end of a character code constant.

29.2 Tokens

token ::=	name token |variable token | integer token | float number token | char code list token | open token | close token | open list token | close list token | open curly token | close curly token | head tail separator token | comma token | end token
name ::=	***<layout text sequence>*** , <name token>
variable ::=	***<layout text sequence>*** , <variable token>
integer ::=	<layout text sequence> , <integer token>
float number ::=	<layout text sequence> , float number token
char code list ::=	<layout text sequence> , char code list token
open ::=	<layout text sequence> , open token
open ct ::=	open token
close ::=	<layout text sequence> , close token
open list ::=	<layout text sequence> , open list token
close list ::=	<layout text sequence> , close list token
open curly ::=	<layout text sequence> , open curly token
close curly ::=	<layout text sequence> , close curly token
ht sep ::=	<layout text sequence> , head tail separator token

comma ::=	<layout text sequence> , comma token
end ::=	<layout text sequence> , end token

A token shall not be followed by characters such that concatenating the characters of the token with these characters forms a valid token as specified by the above syntax.

Layout text

Layout text separates tokens and is also used to resolve ambiguities. Layout text determines whether

1. . (dot) is a graphic token or an end token,
2. An atom followed by an open token is a functor or an operator.

layout text sequence ::=	layout text, layout text
layout text ::=	layout char | comment

The comment text of a ***single line comment*** shall not contain a new line char. The comment text of a ***bracketed comment*** shall not contain the comment close sequence.

comment ::=	single line comment | bracketed comment
single line comment ::=	end line comment char, comment text, new line char
bracketed comment ::=	comment open, comment text, comment close
comment open ::=	comment 1 char, comment 2 char
comment close ::=	comment 2 char, comment 1 char
comment text ::=	char
comment 1 char ::=	"/"
comment 2 char ::=	"*"

Names

name token ::= identifier token | graphic token | quoted token | semicolon token | cut token

identifier token ::= small letter char, alpha numeric char

A ***graphic token*** shall not begin with the character sequence comment open. A graphic token shall not be the single character . (dot) when it is followed by a layout character.

graphic token ::= graphic token char {, graphic token char}

graphic token char ::= graphic char | backslash char

A ***quoted token*** consists of the characters denoted by the sequence of single quoted char appearing within the quoted token. If this character sequence forms a valid atom without quotes the quoted token shall denote that atom. A quoted token which contains no single quoted chars is the null atom. A quoted token can be spread over two or more lines by means of continuation escape sequences. A quoted token containing one or more continuation escape sequences denotes the same atom as the quoted token obtained by removing the continuation escape sequences from the original quoted token.

quoted token ::= single quote char, single quoted item , single quote char

single quoted item ::= single quoted char | continuation escape sequence

continuation escape sequence ::= backslash char , new line char

’abc’ and abc denote the same atom. But ’\ \ /’ and \ \ / do not denote the same atom because \ is used to start an escape sequence in a quoted token.

semicolon token ::= semicolon char

cut token ::= cut char

Quoted characters

A ***quoted char*** can be a single quoted char or a double quoted char or a back quoted char. A quoted char which consists of a graphic char, or an alpha numeric char, or a solo char, or a space char denotes that char.

single quoted char ::= non quote char | single quote char , single quote char | double quote char | back quote char

A single quoted char which consists of two adjacent single quote chars denotes a single quote char.

double quoted char ::= non quote char | single quote char | double quote char , double quote char | back quote char

A double quoted char which consists of two adjacent double quote chars denotes a double quote char.

back quoted char ::= non quote char | single quote char | double quote char | back quote char , back quote char

A back quoted char which consists of two adjacent back quote chars denotes a back quote char.

non quote char ::= graphic char | alpha numeric char | solo char | space char | meta escape sequence | control escape sequence | octal escape sequence | hexadecimal escape sequence

A ***meta escape sequence*** denotes the escaped meta char.

meta escape sequence ::= backslash char , meta char

A ***control escape sequence*** denotes the control character indicated by the name of the symbolic control char.

control escape sequence ::= backslash char , symbolic control char

symbolic control char ::= symbolic alert char | symbolic vertical tab char | symbolic horizontal tab char | symbolic backspace char | symbolic form feed char | symbolic new line char | symbolic carriage return char symbolic alert char ::= "a"

symbolic backspace char ::= "b"

symbolic form feed char ::= "f"

symbolic new line char :: = "n"

symbolic carriage return char ::= "r"

symbolic horizontal tab char ::= "t"

symbolic vertical tab char ::= "v"

symbolic hexadecimal char ::= "x"

An ***octal or hexadecimal escape sequence*** denotes the character from the processor character set ([Ref: processorcharacterset]) whose value according to the collating sequence ([Ref: collatingsequence]) is equal to the value denoted by the octal or hexadecimal constant.

octal escape sequence ::= backslash char, octal digit char, octal digit char, backslash char

hexadecimal escape sequence ::= backslash char, symbolic hexadecimal char, hexadecimal digit char, hexadecimal digit char, backslash char

Numbering the characters of an atom

- The characters of a non-empty atom are numbered from one upwards.
- An empty list consists of the two characters `[]`.
- A curly bracket pair consists of the two characters `{}`.
- In an identifier token the initial small letter char and each alpha numeric char are a single character of the atom.
- In a graphic token each graphic token char is a single character of the atom.

- In a quoted token each single quoted char is a single character of the atom.
- In a cut token the cut char is the first and only character.
- In a semicolon token the semicolon char is the first and only character.

Variables

variable token ::= anonymous variable | named variable

anonymous variable ::= variable indicator char

named variable ::= variable indicator char , alpha numeric char , alpha numeric char | capital letter char , alpha numeric char

variable indicator char ::= underscore char

Integer numbers

integer token ::= integer constant | character code constant | hexadecimal constant

integer constant ::= decimal digit char

integer constant ::= decimal digit char <***integer constant*** >

character code constant

::= zero char , quote char , quoted char

hexadecimal constant ::= hexadecimal constant indicator , hexadecimal digit chars

hexadecimal constant indicator ::= 0x

octal constant ::= octal constant indicator , ocatal digit chars

octal constant indicator = 0o

binary constant ::= binary constant indicator , binary digit chars binary constant indicator = "0b"

An ***integer constant*** is unsigned. Negative integers are defined by the term syntax.

A ***character code constant*** denotes the value of the character according to the collating sequence.

Floating point numbers

float number token ::= integer constant , fraction , [exponent]

fraction ::= decimal point char , decimal digit char , decimal digit char

exponent ::= exponent char , sign , integer constant

sign ::= negative sign char | [positive sign char]

positive sign char ::= "+"

negative sign char ::= "-"

decimal point char ::= "."

exponent char :: = "e" | "E" ;

A ***floating-point constant*** is unsigned. Negative floating-point constants are defined by the term syntax.

Character code lists

A ***char code list token*** denotes the list of collating sequence integers for a sequence of double quoted chars appearing within the char code list token, and can be spread over two or more lines by means of continuation escape sequences. A char code list token containing one or more continuation escape sequences denotes the same list as the char code list token obtained by removing the continuation escape sequences from the original double quoted token.

char code list token ::= double quote char , double quoted item, double quote char

double quoted item ::= double quoted char | continuation escape sequence

Other tokens

open token ::=	open char
close token ::=	close char
open list token ::=	open list char
close list token ::=	close list char
open curly token ::=	open curly char
close curly token ::=	close curly char
head tail separator token ::=	head tail separator char
comma token ::=	comma char

A , (comma) has three different meanings, depending on the context where it appears:

- it can separate arguments of a compound term.
- it can separate elements of a list.
- or can be an operator.

end token ::=	end char
end char ::=	"."

A term shall be terminated by . (end char). An end char shall not be followed by any character other than layout text, that is a layout character or a %.

Collating sequence

The collating sequence is defined implicitly by associating a unique collating sequence integer with each character.

- The collating sequence integer for an unquoted character is defined as the integer value of the byte corresponding to that character in the code table of

IS8859 and an implementation defined value for the new line char.

- The collating sequence integer for a quoted character which is not a control escape sequence or a binary escape sequence or an octal escape sequence or a hexadecimal escape sequence is the collating sequence integer for the unquoted character that the quoted character denotes.
- The collating sequence integer for a quoted character which is a control escape sequence is implementation defined .
- The collating sequence integer for a quoted character which is a binary escape sequence is the value of the binary characters interpreted as an binary integer.
- The collating sequence integer for a quoted character which is an octal escape sequence is the value of the octal characters interpreted as an octal integer.
- The collating sequence integer for a quoted character which is a hexadecimal escape sequence is the value of the hexadecimal characters interpreted as a hexadecimal integer.
- The collating sequence integer for each extended character shall also be implementation defined .

29.3 Prolog Text and Data

Prolog text is a sequence of read-terms which denote directives and the clauses of predicates.

Prolog text

Prolog text is a sequence of directives and clauses.

```
prolog text ::=          directive, prolog text

prolog text ::=          clause, prolog text

prolog text ::=
```

Directives

directive ::=	directive term, end
directive term ::=	term

Clauses

clause ::=	clause term, end
clause term ::=	term

Condition: The principal functor of clause term cannot be :-/1.

Data

A Prolog ***read-term*** can be read as data by calling the predicate read_term/3.

read term ::=	term, end

Any layout text before the term is ignored. A read-term ends with the end token.

29.4 Terms

Every Prolog term is either a constant, a variable, or a compound term. Each Prolog term is assigned a priority by the syntax rules. Unless explicitly noted in the following, the default priority of 0 is assigned.

term ::=	constant | variable | compound term
consttant ::=	number | negative number | atom
number ::=	integer | float number
negative number ::=	- number

The prefix operator - with a numeric constant as operand denotes the corresponding negative constant.

Atoms

An atom which is an operator shall not be the immediate operand of an operator. The priority of a term consisting of an operator is therefore given the priority 1201 rather than the normal 0. An atom which is an operator shall be bracketed in order to denote a term.

An ***atom*** can be a name, the empty list, or a curly bracket pair.

atom ::=	name | empty list | curly bracket pair
empty list ::=	open list, close list
curly bracket pair ::=	open curly, close curly

Variables

Variables have priority 0

Compound terms - Functional Notation and Expressions

compound term ::=	functional notation term | operator notation term

Every ***compound term*** can be expressed in functional notation. When the principal functor is an operator, it can also be expressed in ***operator notation***. When the principal functor is ./2 it can also be expressed as a ***list***, and sometimes it can be expressed as a ***character code list*** . When the principal functor is {}/1 it can also be expressed as a ***curly bracketed expression***.

Compound Terms - Functional Notation

Functional notation is a subset of the Prolog syntax in which all compound terms can be expressed. Perentheses are needed around an atom which is defined as an operator when it is an operand.

A compound term written in functional notation has the form

$$f(A_1,...,A_n)$$

where the arguments A_i are separated by , (comma). Each argument is an ***expres-***

sion.

functional notation term ::=atom, open ct, arg list, close

arg list ::= exp

arg list ::= exp, comma, arg list

Expressions

An ***expression*** (represented by exp in the syntax rules) occurs as the argument of a compound term or element of a list. It can be an atom which is an operator, or a term with priority less than 999. When an expression is an arbitrary term, its priority is less than the priority of the , (comma) operator so that there is no conflict between comma as an infix operator and comma as an argument or list element separator.

exp ::= atom

Condition: If atom is an operator, it is not a comma.

exp = term

Condition: The priority of term is $p \leq 999$.

This concept of an "expression" ensures that both the terms f(x,y) and f(:-, ;, [:-, :-|:-]) are syntactically valid whatever operator definitions are currently defined. Comma is special, and the following terms are syntax errors: f(,,a), [a,,|v], and [a,b|,].

Compound terms – operator notation

Operator notation can be used for writing compound terms whose functor symbol is an operator. There are some predefined operators, and the programmer can define others with op/3. An operator is defined by its name, specifier and priority. The name of an operator is a name or a comma. The priority of an operator is an integer in the range 1-1200. The lower the priority the stronger binds the operator. The specifier of an operator (defined by the table below) is a mnemonic that defines the class (prefix, infix or postfix) and the associativity (right, left or nonassociative) of the operator.

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

Table 7: Operator Specifiers

- An operand with the same (or smaller) priority as a right associative operator which follows that operator need not be bracketed.
- An operand with smaller priority than a left associative operator which precedes that operator need not be bracketed.
- An operand with the same priority as a left associative operator which precedes that operator need only be bracketed if the principal functor of the operand is a right associative operator.
- An operand with the same priority as a non-associative operator must be bracketed.

The lterm non-terminal denotes a subset of terms, namely those allowed as the left operand of a left associative operator with a given priority. The lterm non-terminal is introduced to assign an unambiguous reading to expressions such as fy t1 yf where the operators have the same priority.

29.4.1 Operand

```
term  = lterm ;
```

Abstract:	*a*	*a*
Priority:	*n*	*n*

lterm = term ;

Abstract:	*a*	*a*
Priority:	*n*	*n*-1

A term with smaller priority can always occur where a term of larger priority is allowed.

term = open, term, close ;

Abstract:	*a*	*a*
Priority:	0	1201

term = open ct, term, close ;

Abstract:	*a*	*a*
Priority:	0	1201

Brackets are used to override the priority of operators.

Operators as functors

	lterm	::=	term,	op,	term
Priority:	*n*		*n*-1	*n*	*n*-1
Specifier:				xfx	

	lterm	::=	lterm,	op,	term
Priority:	*n*		*n*	*n*	*n*-1
Specifier:				`yfx`	

	lterm	::=	term,	op,	term
Priority:	*n*		*n*-1	*n*	*n*
Specifier:				`xfy`	

	lterm	::=	lterm,	op
Priority:	*n*		*n*	*n*
Specifier:				`yf`

	lterm ::=	term,	op
Priority:	*n*	*n*-1	*n*
Specifier:			`xf`

	lterm ::=	op,	term
Priority:	*n*	*n*	*n*
Specifier:		`fy`	

Condition: If `term` is a numeric constant, `op` is not `-` .

Condition: The first token of `term` is not `open ct .`

	lterm ::=	op,	term
Priority:	*n*	*n*	*n*-1
Specifier:		`fx`	

Condition: If `term` is a numeric constant, `op` is not `-`

Condition: The first token of `term` is not `open ct`

The last condition defines the use of - in the term -(1,2) as functor and the use in - (1,2) as prefix operator.

Operators

An ***operator*** is a name or a comma which is either a predefined operator or a ***defined operator*** created using the builtin op/3. The priority assigned to the operator is that given by the table of predefined operators, is 1000 for comma, and is the value of the first argument of the call on op/3 when for defined operators.

There shall not be two operators with the same class and name. There shall not be an infix and a postfix operator with the same name.

Note that comma is a solo character, and a token, but not an atom.

`','` is a predefined operator and is equivalent to comma.

The third argument of op/3 is any name except `','`, `'[]'`, and `'{}'`, so the precedence of the comma operator cannot be changed, and so empty lists and curly bracket pairs cannot be treated as operators.

The constraints on multiple operators allow a parser to decide immediately the specifier of an operator without too much look ahead. For example

t1 yf_or_yfx fy_or_yf t2 =   t1 yf_or_yfx (fy_or_yf t2)

t1 yf_or_yfx fy_or_yf yf   =   ((t1 yf_or_yfx) fy_or_yf) yf

In these cases knowledge about the complete term is necessary in order to decide whether to interpret the yf_or_yfx as a yf or yfx operator.

Predefined operators

The table below defines the predefined operators which are initially recognized by a ISO Prolog standard conforming processor.

Priority	Specifier	Operator(s)
1200	xfx	:-
1200	fx	:- ?-
1150	fx	dynamic multifile discontiguous
1100	xfy	;
1050	xfy	->
1000	xfy	,
700	xfx	= \=
700	xfx	== \== @< @=< @> @>=
700	xfx	=..
700	xfx	is =:= =\= < =< > >=
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfx	**
200	xfy	^

Priority	Specifier	Operator(s)
200	fy	- \

The ***predicates*** defined to be operators are:

(a) Term unification,

(b) Term comparison,

(c) =../2 (univ),

(d) Arithmetic evaluation

(e) ^ /2.

The ***control constructs*** defined to be operators are:

(a) ,/2 (conjunction),

(b) ;/2 (disjunction),

(c) ->/2 (if-then).

The ***evaluable functors*** defined to be operators are:

(a) binary arithmetic functors,

(b) -/1 (negation),

(c) logical functors.

Compound terms – list notation

A ***list*** is either the empty list or a non-empty list, which is a compound term with principal functor ./2 (dot).

```
list   ::=          open list, close list | open list, items, close list

items  :: =         exp, comma, items | exp, ht sep, exp  | exp
```

A list is generally of the form

[E1,...,En | Tail]

where the items are separated by , (comma).

Examples

The following examples show terms expressed in list and functional notation:

[a] == .(a, []).

[a, b] == .(a, .(b, [])).

[a | b] == .(a, b).

Compound terms – curly bracket notation

A (compound) term can also be a ***curly bracket pair*** or a ***curly bracketed expression***, which is a compound term with principal functor `{}/1`.

curly bracketed expression ::=open curly, term, close curly

Examples

The following examples show terms expressed in curly bracket and functional notation.

{a} == {}(a).

{a, b, c} == {}(‘,’(a,’,’(b,c))).

Compound terms – character code list notation

A ***character code list*** is either the empty list or a non-empty list, which is a compound term with principal functor ./2 (dot).

term = char code list ;

Abstract: *l* *l*

Priority: 0

Builtins

30 Builtins Reference

This section documents each of the builtin predicates provided with ALS Prolog. The ALS Builtin Predicates Reference Section is similar in format to the Unix man pages. Each predicate in ALS Prolog is documented here in alphabetical order. An index to the builtins appears at the very end of this section. A complete list of all predicates - builtins, foreign interface, library - is provided in the Master Index of Predicates.

30.1 Reference Page Format

The header of each page of the reference contains the following information

- Prolog predicate name
- category of predicate

After the header, a number of paragraphs appear, the first of which is the **NAME** paragraph. Not all of the paragraphs appear for every builtin predicate, but **NAME**, **FORMS** , and **DESCRIPTION** always appear. The following is a list of the paragraphs, and their associated purposes:

- **NAME** — states the name of the Prolog predicate, together with what it does.
- **FORMS** — illustrates the form of a call to the Prolog predicate.
- **DESCRIPTION** — provides a description of what the Prolog predicate does. The description goes into more detail than the information given in the **NAME** paragraph.
- **EXAMPLES** — provides some examples of use of the predicate being used.
- **NOTES** — describes unusual or surprising details that should be noted by the reader, and also describes any differences between versions of ALS Prolog.
- **SEE ALSO** — provides references to other information.
- **ERRORS** — describes possible error conditions and actions.

30.2 Naming Conventions

The ALS Prolog naming convention for the builtin Prolog predicates is that the names of the predicates consist entirely of lowercase alphabetical characters. The exceptions to this rule are the undocumented predicates listed at the end of this reference section.

NAME

`!` – removes choice points

FORMS

```
FirstGoal, !, SecondGoal
FirstGoal, !; SecondGoal
```

DESCRIPTION

Discards all choice points made since the parent goal started execution, including the choice points, if any, created by calling the parent goal. In the following two cases, a cut in `Condition` will remove all choice points created by the `Condition`, any subgoals to the left of the `Condition`, and the choice point for the parent goal.

```
Condition = (Things, !, MoreThings)

Condition->TrueGoal; FalseGoal
call(Condition)
```

In other words,

```
->
call/1
;
:
,
```

are transparent to cut. The ISO Prolog Standard requires that `call/1` be opaque to cut. At this time, ALS Prolog deviates from the standard.

EXAMPLES

In the following example, the solution `eats(chris,pizza)`causes a cut to be executed. This removes the choice point for the goal `eats/2`. As a result, the solution `eats(mick,pizza)` is not found, even though Mick will eat anything.

```
?- [user].
Consulting user.
  eats(chris,pizza) :- !.
  eats(mick,Anything).
  user consulted.

yes.
?- eats(Person,pizza).
Person = chris;

no.
```

The next example shows that `not/1` is opaque to cut. This means that a '`!`' inside the call to `not/1` will not cut out the choicepoint for `not/2`, or any other choicepoints created by goals to the left of `not/2`.

```
?- not((!,fail)).

yes.
```

Notice the extra pair of parentheses above. This is to prevent the parser from creating a goal to `not/2` instead of `not/1`. In the next example, the transparency of `call/1` with respect to cut is shown:

```
?- [user].
Consulting user.
  cool(peewee) :- call((!,fail)).
  cool(X).
  user consulted.

yes.
?- cool(peewee).

no.
```

```
?- cool(bugsbunny).

yes.
```

`peewee` is not cool because the '`!`' removed the choicepoint for `cool/1`. The `fail` after the '`!`' prevented `cool/1` from succeeding. The rationale for having cut behave this way is so that:

```
cool(peewee) :- call((!,fail)).
```

will be equivalent to

```
cool(peewee) :- !,fail.
```

The next example shows the transparency of `->` with respect to cut.

```
?- [-user].
Reconsulting user.
  cool(X) :- (X=peewee,!) -> fail.
  cool(X).
  user reconsulted.

yes.
?- cool(peewee).

no.
```

Again, `peewee` is not considered cool. In the goal

```
?- cool(peewee).
```

the '`!`' after `X=peewee` cuts the choicepoint for `cool/1`. The condition succeeds, causing `fail` to be executed. However, the second clause is never reached because the choicepoint has been cut away. Consequently, the goal fails. The goal

```
?- cool(daffyduck).
```

succeeds because the '`!`' is never reached in the condition of `->`. The `->` fails because there is no `else` subgoal. This causes the next clause for `cool/1` to be executed. This clause always succeeds, therefore `daffyduck` is considered cool.

SEE ALSO

`->/2, not/1,`

[Bowen 91, 7.1], [Sterling 86, 11], [Bratko 86, 5.1], [Clocksin 81, 4.2].

NAME

`,/2` – conjunction of two goals

FORMS

`FirstGoal,SecondGoal`

DESCRIPTION

The first argument is called as a goal. If it succeeds, then the `SecondGoal` will be run. If either goal fails, the most recent alternative will be attempted after backtracking.

EXAMPLES

The following example shows the use of the '`,`' connector:

```
?- [user].
Consulting user.
  lucky(mick,love).
  boss(mick,jerri).
  user consulted.

yes.
?- lucky(Who,What), boss(Who,Boss).

Who = mick
What = love
Boss = jerri

yes.
```

The goal submitted to the Prolog shell consists of two subgoals

- lucky(Who,What)
- boss(Who,Boss)

The subgoals are connected together by using the '`,`' predicate. In the next example, the first subgoal fails, so the second subgoal is not executed:

```
?- fail, write('Help, I''m stuck in an example'), nl.

no.
```

The following shows that `,` works the same in `call/1`:

```
?- call((fail, write('Help, I''m stuck in an
example'), nl)).

no.
```

Note that the parentheses around the argument to `call/1` are to keep the parser from creating a call to `call/3`.

SEE ALSO

`call/1, :/2, ;/2,`

[Bratko 86, 2.3], [Clocksin 81, 6.7].

NAME

`->/2` – if-then, and if-then-else

FORMS

```
Condition -> TrueGoal
Condition -> TrueGoal ; FalseGoal
```

DESCRIPTION

If `Condition` succeeds, then `TrueGoal` will be executed. If-then implicitly cuts the `Condition`. Cuts that occur within `Condition` or `TrueGoal` will cut back to the head of the parent clause. If `Condition` fails, then the call to `->/2` fails. The second form, results from the interaction between `;/2` and `->/2` because

```
       Condition -> TrueGoal ; FalseGoal
```

is actually executed as:

```
       (Condition -> TrueGoal) ; FalseGoal
```

In this case, `FalseGoal` will be executed instead of `TrueGoal` when `Condition` fails.

Cuts occurring in

- Condition
- TrueGoal
- FalseGoal

cut back to the head of the parent clause.

EXAMPLES

```
?- true -> write(a).
a
yes.
?- fail -> write(a).

no.
```

```
?- fail -> write(a) ; write(b).
b
yes.
```

SEE ALSO

`!/0`, `not/1`,

[Bowen 91, 7.1], [Sterling 86, 11], [Bratko 86, 5.1], [Clocksin 81, 4.2].

NAME

`:/2` – calls a goal in the specified module

FORMS

```
Module:Goal
```

DESCRIPTION

`Module` should be instantiated to a name of a module and `Goal` should be instantiated to a non-variable term. The `Goal` will be executed as if the call to `Goal` appeared in the module named `Module`. Any module dependent procedures such as `assert/1` or `retract/1` will operate on the procedures defined in `Module`. Any cuts appearing in `Goal` will cut back through the head of the clause that contained the call to `:/2`.

EXAMPLES

```
builtins:write(foobar)
behavior:setof(Person, eats_linguini(Person),
Persons)
```

ERRORS

If `Module` is unbound or not bound to a valid module name, the call to `:/2` will fail.

SEE ALSO

`call/1,  User Guide (Modules).`

NAME

`;/2` – disjunction of two goals

FORMS

```
FirstGoal ; SecondGoal
```

DESCRIPTION

The `FirstGoal` is called. Later, upon backtracking, `SecondGoal` will be called. Cuts appearing in either `FirstGoal` or `SecondGoal` will cut back to the head of the clause which contained the call to `;/2`.

EXAMPLES

The following example shows the use of '`;`' as the boolean or connective:

```
?- [user].
Consulting user.
  language(postscript).
  language(pascal).
  food(burrito).
  food(crab).
  food(steak).
  user consulted.

yes.
?- language(postscript) ; food(postscript).

yes.
```

Notice that although `postscript` isn't a food, the goal succeeds. This is because only one of the two subgoals has to succeed for '`;`' to succeed. In the next example, we add a few more facts to the database. This example shows that '`;`' goal also succeeds if both of its arguments can succeed.

```
?- [user].
Consulting user.
  food(prolog).
  language(prolog).
  user consulted.

yes.
?- language(prolog) ; food(prolog).

yes.
```

Note that the `food(prolog)`goal is never run, even though it is `true`. The next example shows that '`;`' will fail if neither of the subgoals succeed.

```
?- language(fortran) ; food(fortran).

no.
```

The next example illustrates the behavior of '`;`' upon backtracking. The semicolons after the shown answers are typed in by the user interactively:

```
?- language(X) ; food(X).

X = postscript;
X = pascal;
X = prolog;
X = burrito;
X = crab;
X = steak;
X = prolog;

no.
```

SEE ALSO

`!/0, ->/2,`

[Bowen 91, 7.1], [Sterling 86, 10.4], [Bratko 86, 2.3], [Clocksin 81, 6.7].

NAME

`</2`	– The left expression is less than the right expression
`>/2`	– The left expression is greater than the right expression
`=:=/2`	– The left and right expressions are equal
`=\=/2`	– The left and right expressions are not equal
`=</2`	– The left expression is less than or equal to the right
`>=/2`	– The left expression is greater than or equal to the right

FORMS

```
Expression1  <        Expression2
Expression1  >        Expression2
Expression1 =:=       Expression2
Expression1 =\=       Expression2
Expression1 =<        Expression2
Expression1 >=        Expression2
```

DESCRIPTION

Both arguments to each relational operator should be instantiated to expressions which can be evaluated by `is/2`. The relational operator succeeds if the relation holds for the value of the two arguments, and fails otherwise. A relational operator will fail if one or both of its arguments cannot be evaluated.

EXAMPLES

```
?- -7*0 =< 1+1.

yes.

?- 1+1 =< 7*0.
```

```
no.
```

ERRORS

The ISO Prolog Standard requires that a calculation error be thrown when the arguments cannot be evaluated a for these operators. At this time, ALS Prolog does not conform to this requirement.

NAME

`=/2`	– unify two terms
`\=/2`	– test if two items are non-unifiable

FORMS

```
Arg1  = Arg2
Arg1 \= Arg2
```

DESCRIPTION

The procedure `=/2` calls the Prolog unifier to unify `Arg1` and `Arg2`, binding variables if necessary. The occurs check is not performed. `=/2` is defined as if by the clause:

Term = Term.

If the two terms cannot be unified, `=` fails. The procedure `\=` succeeds if the two terms cannot be unified. This is different than the `=\=` and the `\==` procedures.

EXAMPLES

The following examples illustrate the use of `=` .

```
?- f(A,A) = f(a,B).
A = a
B = a

yes.
?- f(A,A) = f(a,b).

no.
?- X = f(X).
X = f(f(f(f(f(f(f(f(f(f(f(...)))))))))))
?- X \= 1.

no.
```

```
?- X \== 1.
X = _3

yes.
```

Note that in the next to last example, the depth of the printing is much deeper than shown here.

SEE ALSO

`==/2,\==/2,eq,noneq,`

[Bowen 91, 4.6], [Bratko 86, 2.7], [Clocksin 81, 6.8], [Sterling 86, 4.1].

NAME

`=../2` – translates between lists and terms

FORMS

```
Term  =.. List
```

DESCRIPTION

Either `Term` or `List` must be instantiated to a non-variable term. When `Term` is instantiated to a constant, `List` will be unified with the singleton list whose element is `Term`. If `Term` is a structured term, then `List` will be unified with a list whose head is the principal functor of `Term` and whose tail is the list of arguments in `Term`.

When `List` is instantiated to a singleton list, whose element is an atom, then `Term` will be unified with the element. If `List` has at least two elements, the first of which is an atom, then `Term` will be unified with a term whose principal functor is the head of `List` and whose arguments are the remaining elements of `List` in order. This predicate is commonly known as `univ`.

EXAMPLES

```
?- my_pred(tom,A+B,f(X)) =..[Functor|Args].
Functor = my_pred
Args = [tom,A+B,f(X)]
A = _2
B = _4
X = _8

yes.
?- X =.. [-,3,1].
X = 3-1

yes.
?- 1 =.. [1].
```

```
yes.
```

ERRORS

`=../2` fails if:

1 Neither `Term` nor `List` is instantiated
2 `List` is not a proper list whose head is an atom
3 `List` is unistantiated and `Term` is not an atom or a structure.

NOTES

The ISO Prolog Standard requires that thes above error be thrown when the arguments cannot be evaluated a for these operators. At this time, ALS Prolog does not conform to this requirement.

SEE ALSO

`functor/3, arg/3,`

[Bowen 91, 7.6], [Bratko 86, 7.2], [Sterling 86, 9.2].

NAME

`==/2`	– terms are identical
`\==/2`	– terms are not identical

FORMS

```
Term1  == Term2
Term1 \== Term2
```

DESCRIPTION

`Term1` is identical to `Term2` (`Term1 == Term2`) if they can be unified, and variables occupying equivalent positions in both terms are identical. For atoms and variables, this is an absolute identity check. Viewing Prolog terms as trees in memory, `==/2` determines whether `Term1` and `Term2` are isomorphic trees whose leaves are identical. Unlike `=/2`, no variables are bound inside a call to `==/2.` \== fails when == succeeds, and conversely.

EXAMPLES

```
?- bar \== foo.

yes.
?- f(b) == f(b).

yes.
?- X == Y.

no.
?- f(X) \== f(X).

no.
?- [a,b,c] \== [a,b,c].

no.
```

SEE ALSO

[Bowen 91, 7.4], [Clocksin 81, 6.8], [Bratko 86, 3.4].

NAME

`@=</2`	– The left argument is not after the right argument
`@>=/2`	– The left argument is not before the right argument
`@</2`	– The left argument is before the right argument
`@>/2`	– The left argument is after the right argument

FORMS

```
Arg1 @=< Arg2
Arg1 @>= Arg2
Arg1 @<  Arg2
Arg1 @>  Arg2
```

DESCRIPTION

These predicates compare two terms according to the standard order as defined by compare/3. The terms are compared as-is without any transformation or interpretation. The order of a partially instantiated term may change as the level of instantiation changes..

EXAMPLES

```
?- foobar(something) @> foobar.

yes.
?- cement @> mortar.

no.
?- inflation_today @=< inflation_tomorrow.

yes.
?- rain_in(newYork) @>= rain_in(arizona).

yes.
```

SEE ALSO

`compare/3`.

NAME

`abolish/2` – remove a procedure from the database

FORMS

```
abolish(Name,Arity)
```

DESCRIPTION

All the clauses for the specified procedure `Name/Arity` in the current module are removed from the database. The module from which to abolish clauses can be specified with a '`:`'. Given appropriate arguments, `abolish/2` will succeed whether or not it actually removed any clauses.

EXAMPLES

```
?- abolish(zip,3).

yes.
?- othermodule:abolish(victim,7).

yes.
```

ERRORS

If `Name` is not an atom or `Arity` is not an integer, `abolish/2` fails.

SEE ALSO

:/2.

NAME

`abort/0` – return execution immediately to the Prolog shell

FORMS

```
abort
```

DESCRIPTION

The current computation is discarded and control returns to the Prolog shell.

EXAMPLES

```
error(Message) :-
    write(Message), nl,
    abort.
```

SEE ALSO

`see/1`, `tell/1`.

NOTES

If ALS Prolog was started with the `-b` command line option, the system exits when `abort` is executed.

NAME

`'$access'/2` – determine accessability of a file

FORMS

```
'$access'(FileName,AccessMode)
```

DESCRIPTION

`FileName` can be a symbol, a UIA, or a list of ASCII characters. `'$access'/2` checks whether or not the given file is accessible with the given mode, where `AccessMode` should be one of the following:

```
4 : write
2 : read
0 : existence
```

These access modes are used as indicated on Unix systems and the Macintosh. However, they are ignored on DOS systems. On DOS systems, the file is only checked as to whether or not it is read accessible.

EXAMPLES

```
?- '$access'('../foo/foobar.pl',2).

yes.
```

NAME

`als_system/1`	– Provides system environmental information.
`sys_env/3`	– Provides brief system environmental information.

FORMS

```
als_system(INFO_LIST)
sys_env(OS,OS_Variation,Processor)
```

DESCRIPTION

Portable programs which interact with the operating system must take account of the variations in the system environment. `als_system/1` provides a method of achieving this goal. When the ALS Prolog system initializes itself, the underlying C substrate asserts a single fact

```
als_system(INFO_LIST)
```

in the module `builtins` in the Prolog database. The argument of this fact is a list of equations of the form

```
property = value
```

Each property appears at most once. The properties and their possible values are listed in the table below.

Property Tag	Value Examples
`os`	`unix,dos,macos,mswins32,vms`
`os_variation`	`(unix):'solaris2.4', 'sunos4.1.3', hpux9.05,.. (mswin32):mswin95, mswinNT, mswin32s`
`processor`	`port_thread,port_byte, i386,m68k,m88k,sparc,powerpc`
`manufacturer`	`generic,sun,motorola,dec,`

Property Tag	Value Examples
`prologVersion`	`’nnn-mm’`
`wins`	`nowins,motif,macos,...`

For most purposes, knowing the operating system (`OS`), and possibly the `Processor`, is what matters. Consequently, another small fact,

```
sys_env(OS,OS_Variation,Processor)
```

is asserted during initialization, recording the values of the `os`, the `os_variation`, and the `processor` properties from the `als_system` list description.

EXAMPLES

On a Sun SPARC running Solaris 2.4, TTY portable version:

```
?- als_system(X).

X = [os = unix,os_variation = ‘solaris2.4’,
processor = port_thread,manufacturer =
generic,prologVersion = ‘1-76’,wins = nowins]
```

NAME

`append/3`	– append two lists
`dappend/3`	– append two lists

FORMS

```
append(List1,List2,List3)
dappend(List1,List2,List3)
```

DESCRIPTION

`List3` is the result of appending `List2` to the end of `List1`. `dappend` is the determinate version of `append`.

EXAMPLES

```
?- append([a,b],[c,d],E).
E = [a,b,c,d]

yes.
?- append([a,b],[C,D],[a,b,c,d]).
C = c
D = d

yes.
```

NOTES

`append` and `dappend` are defined by:

```
append([],L,L).
append([H|T],L,[H|TL]) :- append(T,L,TL).

dappend([],L,L) :- !.
dappend([H|T],L,[H|TL]) :- dappend(T,L,TL).
```

NAME

`arg/3` – access the arguments of a structured term

FORMS

`arg(Nth,Structure,Argument)`

DESCRIPTION

`Argument` will be unified with the `Nth` argument of `Structure`. `Nth` must be a positive integer. `Structure` should be a compound term whose arity is greater than or equal to `Nth`. When these conditions hold, `Argument` will be unified with the `Nth` argument of `Structure`.

EXAMPLES

```
?- arg(2,stooges(larry,moe,curly),X).
X = moe

yes.
?- arg(2, [a,b,c],X).
X = [b,c]

yes.
```

ERRORS

If `Nth` is not an integer greater than `0` and less than or equal to the arity of `Structure`, `arg/3` will fail. `Structure` must be instantiated to a structured term.

SEE ALSO

`functor/3`, `=../2`,

[Bowen 91, 7.6], [Sterling 86, 9.2], [Bratko 86, 7.2], [Clocksin 81, 6.5]

NAME

`assert/1`	– adds a clause to a procedure
`assert/2`	– adds a clause to a procedure
`asserta/1`	– adds a clause at the beginning of a procedure
`asserta/2`	– adds a clause at the beginning of a procedure
`assertz/1`	– adds a clause at the end of a procedure
`assertz/2`	– adds a clause at the end of a procedure

FORMS

```
assert(Clause)
assert(Clause,Ref)
asserta(Clause)
asserta(Clause,Ref)
assertz(Clause)
assertz(Clause,Ref)
```

DESCRIPTION

The `Clause` is added to the procedure with the same name and arity in the module that `assert` is called from. All uninstantiated variables are re-quantified in the clause before it is added to the database, thus breaking any connection between the original variables and those occurring in the clause in the database. Because of this behavior, the order of calls to assert is important. For example, assuming no clauses already exist for `p/1`, the first one of the following goals will fail, while the second succeeds.

```
?- X = a, assert(p(X)), p(b).

no.
?- assert(p(X)), X = a, p(b).
X = a

yes.
```

The placement of a clause by `assert/1` is defined by the implementation. `asserta/1` always adds its clause before any other clauses in the same procedure, while `assertz/1` always adds its clause at the end. Each form of assert can take an optional second argument (normally an uninstantiated variable) which is the database reference corresponding to the clause that was added. `:/2` can be used to specify in which module the assert should take place. The database reference argument is normally passed as an uninstantiated variable.

EXAMPLES

The following example shows how the different forms of `assert` work:

```
?- assert(p(a)), asserta(p(c)), assertz(p(b)).

yes.
?- listing(p/1).
% user:p/1
p(c).
p(a).
p(b).

yes.
```

Notice that the order of the clauses in the database is different than the order in which they were asserted. This is because the second assert was done with `asserta/1`, and the third assert was done with `assertz/1`. The `asserta/1` call put the `p(c)` clause ahead of `p(a)` in the database. The `assertz/1` call put `p(b)`at the end of the `p/1` procedure, which happens to be after the `p(a)` clause. The next example demonstrates the use of parentheses in asserting a rule into the Prolog database:

```
?- assertz((magic(X):- wizard(X);
pointGuard(X,lakers))).
X = _1

yes.
?- listing(magic/1).
% user:magic/1
```

```
magic(_24) :-
     wizard(_24)
  ;  pointGuard(_24,lakers).

yes.
```

If the extra parentheses were not present, the Prolog parser would print the following error message:

```
assertz(magic(X) :- wizard(X) ;
pointGuard(X,lakers)).
                 ^
Syntax Error:Comma or right paren expected in
argument list.
```

The next example shows how the `assert` predicates can be used with modules. The first goal fails because there is no module named `animals`. After the module is created, the assertion is successful as you can see by looking at the listing of the `animals` module.

```
?- animals:assert(beast(prolog)).

no.
?- [user].
Consulting user.
  module animals.
  endmod.
  user consulted.

yes.
?- animals:assert(beast(prolog)).

yes.
?- listing(animals:_).
% animals:beast/1
```

```
beast(prolog).

yes.
```

The following example shows the effects of adding clauses to procedures which are part of the current goal:

```
?- assert(movie(jaws)), movie(X),
assert(movie(jaws2)).
X = jaws;

no.
```

The reason this didn't work is an implementation issue. The following is the sequence of events illustrating what happened:

- First the `assert(movie(jaws))` subgoal was run, causing a new procedure to be placed in the Prolog database.
- When the subgoal `movie(X)` was run, no choice point was created because there were no other clauses to try if failure occurred.
- After `movie(X)` succeeded, the second clause of `movie/1` was asserted, and the initial goal succeeded, binding `X` to `jaws`.
- Backtracking was initiated by the ';' response to the solution, but no second solution was found for `movie/1`, even though there was a solution to be found. This was because there was no choice point to return to in `movie/1`.

One of the interesting (and possibly bad) parts to this phenomenon is that the second time this goal is run it will backtrack through the clauses of `movie/1`. This is shown below:

```
?- listing(movie/1).
% user:movie/1
movie(jaws).
movie(jaws2).

yes.
```

```
?- assert(movie(jaws)), movie(X),
assert(movie(jaws2)).
X = jaws;
X = jaws2;
X = jaws;
X = jaws2;
X = jaws2;
X = jaws2

yes.
```

The reason for this, is that there was more than one clause for `movie/1` in the database this time, so a choicepoint was created for the `movie(X)` subgoal. Incidentally, this goal would continue finding the

```
X = jaws2
```

solution. This is because every time the `movie(X)` finds a new solution, it succeeds, thus causing the

```
assert(movie(jaws2))
```

subgoal to run. This adds another clause to the database to be tried when the user causes backtracking by pressing semicolon (;). If you look at the conversation with the Prolog shell shown above, you will notice that the last solution was accepted because no ‘`;`’ was typed after it.

ERRORS

Clauses must be either structured terms or atoms. If clause is a rule, with a principal functor of `:-/2`, then the head and all the subgoals of the clause must either be atoms or structured terms.

NOTES

ALS Prolog provides a global variable mechanism separate from the Prolog database. Using global variables is much more efficient than using `assert` and `retract`.

SEE ALSO

:/2,

[Bowen 91, 7.3], [Clocksin 81, 6.4], [Bratko 86, 7.4], [Sterling 86, 12.2].

NAME

`at_end_of_stream/0`– test for end of the curent input stream
`at_end_of_stream/1`– test for end of a specific input stream

FORMS

```
at_end_of_stream
at_end_of_stream(Stream_or_Alias)
```

DESCRIPTION

`at_end_of_stream/0` will succeed when the stream position associated with the current input stream is located at or past the end of stream.

`at_end_of_stream/1` will succeed when the stream position associated with `Stream_or_Alias` is located at or past the end of stream.

ERRORS

`Stream_or_Alias` is a variable.

——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

——> `domain_error(stream_or_alias,Stream_or_Alias)`

`Stream_or_Alias` is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias)`

NOTES

Calling `at_end_of_stream` may cause an input operation to take place. Thus a call to this predicate could block.

At present, the ALS Prolog implementation of `at_end_of_stream` will throw an error when called with an argument which does not represent an input stream.

SEE ALSO

open/4, current_input/1, flush_output/1, *User Guide (Prolog I/O)*

NAME

`atom/1`	– the term is an atom
`atomic/1`	– the term is an atom or a number
`float/1`	– the term is a floating point number
`integer/1`	– the term is an integer
`number/1`	– the term is an integer or a floating point

FORMS

```
atom(Term)
atomic(Term)
float(Term)
integer(Term)
number(Term)
```

DESCRIPTION

Each of these predicates will succeed when its argument is of the proper type, and fail otherwise. `integer/1` and `float/1` examine the only the representation of a number. For instance, the call `integer(2.0)` will succeed because `2.0` is represented internally as an integer. In addition, `float(4294967296)` will succeed because `4294967296` is represented by a floating point value since it is outside the range of the integer representation.

EXAMPLES

The following are examples of the use of the type predicates:

```
?- atom(bomb).

yes.
?- integer(2001).

yes.
?- float(cement).
```

```
no.
```

SEE ALSO

`var/1`, `nonvar/1`,

[Bowen 91, 7.6], [Sterling 86, 9.1], [Bratko 86, 7.1], [Clocksin 81, 6.3].

NAME

`atom_chars/2`	– convert between atoms and the list of characters representing the atom
`atom_codes/2`	– convert between atoms and the list of character codes representing the atom

FORMS

```
atom_chars(Atom,CharList)
atom_codes(Atom,CodeList)
```

DESCRIPTION

`atom_chars(Atom,CharList)` is true if and only if `CharList` is a character list whose elements correspond to the characters of the atom `Atom`.

`atom_codes(Atom,CodeList)` is true if and only if `CodeList` is a character code list whose elements correspond to the character codes of the atom `Atom`.

EXAMPLES

```
?- atom_chars('the cat in',L).

L = [t,h,e,' ',c,a,t,' ',i,n]

yes.
?- atom_chars(A,[t,h,e,' ',h,a,t,'\n']).

A = 'the hat\n'

yes.

?- atom_codes(A,[65,66,67]).

A = 'ABC'
```

```
yes.
?- atom_codes(holiday, L).

L = "holiday"

yes.
```

ERRORS

`Atom` and `CharList` are variables (`atom_chars/2`)
  ——> `instantiation_error.`

`Atom` and `CodeList` are variables (`atom_codes/2`)
  ——> `instantiation_error.`

`Atom` is neither a variable nor an atom
  ——> `type_error(atom,Atom).`

`CharList` is neither a variable nor a list nor a partial list
  ——> `type_error(list,CharList).`

`CodeList` is neither a variable nor a list nor a partial list
  ——> `type_error(list,CodeList).`

`CharList` is a list but there is a sublist `L` of `CharList` whose first element is neither a variable nor a character
  ——> `domain_error(character_list,L).`

`CodeList` is a list but there is a sublist `L` of `CodeList` whose first element is neither a variable nor a character
  ——> `domain_error(character_code_list, L).`

SEE ALSO

```
number_chars/2, number_codes/2, term_chars/2,
term_codes/2.
```

NAME

`atom_concat/3` – append two atoms together to form a third

FORMS

```
atom_concat(Atom1,Atom2,Atom12)
```

DESCRIPTION

If `Atom1` and `Atom2` are bound to atoms, calling `atom_concat/3` will unify `Atom12` with the atom formed by concatenating the characters of `Atom2` to the end of `Atom1`.

If either or both of `Atom1` or `Atom2` are unbound, then `Atom12` must be bound to an atom. `atom_concat/3` will unify `Atom1` and/or `Atom2` to atoms such that concatenating `Atom2` to `Atom1` will form `Atom12`.

`atom_concat/3` is non-determinate when only `Atom12` is instantiated. Upon backtracking `Atom1` and `Atom2` will take on all possible instantiations such that `Atom2` concatenated to `Atom1` will form `Atom12`.

EXAMPLES

```
?- atom_concat(cater,pillar,A).

A = caterpillar

yes.
?- atom_concat(cater,A,caterpillar).

A = pillar

yes.
?- atom_concat(A,B,abc).

A = ''
B = abc;
```

```
A = a
B = bc;

A = ab
B = c;

A = abc
B = '';
no.

?- atom_concat(1,2,A).
error(type_error(atom,1),[builtins:atom_concat(1,2,_
3282)])

Error: Argument of type atom expected instead of 1.
- Goal:          builtins:atom_concat(1,2,_A)
- Throw pattern: error(type_error(atom,1),
                     [builtins:atom_concat(1,2,_A)])
```

ERRORS

`Atom1` and `Atom12` are variables

    ——> instantiation_error.

`Atom2` and `Atom12` are variables

    ——> instantiation_error.

`Atom1` is neither a variable nor an atom

    ——> type_error(atom,Atom1)

`Atom2` is neither a variable nor an atom

    ——> type_error(atom,Atom2)

`Atom12` is neither a variable nor an atom

    ——> type_error(atom,Atom12)

SEE ALSO

`atom_length/2`, `sub_atom/4`, `atom_chars/2`, `atom_codes/2`.

NAME

`atom_length/3` – determine the length of an atom

FORMS

```
atom_length(Atom,Length)
```

DESCRIPTION

`atom_length/2` must have `Atom` bound to an atom. `Length` is unified with the number of characters in the atom.

EXAMPLES

```
?- atom_length(an_atom, L).

L = 7

yes.
?- atom_length('Another atom\n', L).

L = 13

yes.
?- atom_length(9.5, L).
error(type_error(atom,9.5),[builtins:atom_length(9.5
,_3286)])

Error: Argument of type atom expected instead of 9.5.
- Goal:          builtins:atom_length(9.5,_A)
- Throw pattern: error(type_error(atom,9.5),
                       [builtins:atom_length(9.5,_A)])
```

ERRORS

`Atom` is a variable

——> `instantiation_error.`

`Atom` is neither a variable nor an atom

——> `type_error(atom,Atom)`

`Length` is neither a variable nor an integer

——> `type_error(integer,Length)`

SEE ALSO

`atom_concat/3.`

NAME

`bufread/2`	- runs the Prolog parser on a string of text
`bufread/4`	- similiar to bufread/2, giving additional information

FORMS

```
bufread(Buffer,[Structure|Vars])
bufread(Buffer,[Structure|Vars],FullStop,LeftOver)
```

DESCRIPTION

`bufread/2` takes a Prolog string, `Buffer`, and attempts to transform it into a Prolog term. It does this by:

- Reading the first term out of Buffer (trailing characters from `Buffer` are ignored) and unifying it with `Structure`.
- Returning in `Vars` a list of the quoted variable names occurring in the term read.

If an error has occured, the head will be an error message (a Prolog string), and the tail will be the column number where the error is suspected to have occurred. `bufread/4` is the same as `bufread/2`, except that more information is provided. `FullStop` is a flag indicating whether a full stop has been typed. Its value is 1 if there was a full stop, and 0 if there was not. `LeftOver` is the text that is not transformed yet. Although, `bufread/4` only transforms one term at a time, it returns the text it has not transformed yet in the `LeftOver` argument. This text can then be transformed by issuing another `bufread` call, with `LeftOver` given as the `Buffer`.

EXAMPLES

The following example converts the buffer:

```
    "f(abc, Bob) some stuff"
```

to the term `f(abc,_53)`.

```
   ?- bufread("f(abc, Bob) some stuff", [T | Vars]).
```

```
    T = f(abc,_53)
    Vars = ['Bob']
    yes.
```

Observe that the characters `"some stuff"` are discarded by `bufread/2`. This next examples demonstrates how error messages are returned:

```
?- bufread("hello(",[Message|Column]),
printf("\nMessage: %s\nColumn:
%d\n",[Message,Column]).
Message: Non-empty term expected.
Column: 6
Message =
[78,111,110,45,101,109,112,116,121,32,116,101,
114,109,32,101,120,112,101,99,116,101,100,46]
Column = 6
yes.
```

`bufread/2` can be used to convert strings to integers in the following manner:

```
    ?- bufread("123",[Int|_]).
    Int = 123
    yes.
```

In the following example, the term `inside(Where)` was not terminated with a full stop, so `FullStop` is bound to `0`. There is no leftover text to run, so `LeftOver` is bound to the empty list.

```
?-
bufread("inside(Where)",[Term|Vars],FullStop,LeftOve
r).
Term = inside(_38)
Vars = ['Where']
FullStop = 0
LeftOver = []
yes.
```

If you are writing a shell in Prolog using `bufread/4`, you could write a continuation prompt to tell the user of your shell that they must terminate the term with a full stop. In the next example, the term `food(tai)` was terminated by a full stop, so `FullStop` is bound to `1`. This time, there is some leftover text that can be processed, so `LeftOver` is bound to the remaining Prolog string:

```
    " food(indian). food(chinese). "

?- bufread("food(tai). food(indian). food(chinese).
",
                 [Term|Vars],FullStop,Rest),
       printf("Rest = %s\n",[LeftOver]).
LeftOver =  food(indian). food(chinese).
Term = food(tai)
Vars = []
FullStop = 1
LeftOver =
[32,102,111,111,100,40,105,110,100,105,97,110,41,46,
32,
102,111,111,100,40,99,104,105,110,101,115,101,41,46,
32]
yes.
```

If you are writing a shell such as mentioned above, you can use use the `Left-Over` argument to allow multiple goals per line. You do this by continually calling `bufread/4` and checking whether you still have further input to process in `LeftOver`.

CROSSREF

bufwrite	- see sprintf/3

NAME

`'$c_malloc'/2`	– Allocates a C data area using the system `malloc` call
`'$c_free'/1`	– Frees a C data area
`'$c_set'/2`	– Modifies the contents of a C data area or a UIA
`'$c_examine'/2`	– Examines the contents of a C data area or a UIA

FORMS

```
'$c_malloc'(+Size,-Ptr)
'$c_free'(+Ptr)
'$c_set'(+Ptr_or_UIA,+FormatList)
'$c_examine'(+Ptr_or_UIA,+FormatList)
```

DESCRIPTION

The following predicates are C defined builtins in ALS Prolog. `'$c_malloc'/2` allocates a C data area, and `'$c_free'/1` frees it, `'$c_set'/2` is used to modify the contents of a C data area or a UIA, and `'$c_examine'/2` is used to examine the contents of a C data area or a UIA. `'$c_malloc'(Size,Ptr)` is `true` if `Size` is a positive integer and `Ptr` (an integer) unifies with the address of the first byte of a data area allocated by the system call "malloc". The call fails if malloc returns a null pointer. `'$c_free'(Ptr)` is `true` if `Ptr` is a number, and it invokes the system call "free" to free the data area pointed by `Ptr`. `'$c_set'(Ptr_or_UIA,FormatList)` is `true` if `Ptr_or_UIA` is bound to the address of a C data area or a UIA, and `FormatList` is a non-empty list of 3-ary or 4-ary terms of the form

```
f(+Offset,+Type,+Value{,+Length})
```

and the call modifies the contents of the data area as explained below. In each term, `Offset` is the offset of the field from the start address of the data area. `Type` is the "C type" of the field, which is one of the following symbols : `int,long,short,ptr,char,str,float,double`. They have the obvious correspondence with C data types. Value is the data that the field should be set to. If Type is one of int, long, short, ptr, char, float or double, then Value must be a number, otherwise Type is str and Value must be an atom and a null terminated string name of the atom is copied into the receiving data are without overflow checks. Length (optional) must be a number and has meaning only when Type is str, and at most Length bytes are copied into the receiving data area. '$c_examine'(Ptr_or_UIA,FormatList) is true if Ptr_or_UIA is bound to the address of a C data area or a UIA, and FormatList is a nonempty list of 3-ary or 4-ary terms of the form

```
f(+Offset,+Type,-Value{,+Length})
```

whose arguments are interpreted as in '$c_set'/2 (above) except that now a data item of the specified type is extracted from the data area and unified with Value.

NAME

`call/1` – calls a goal

FORMS

```
call(Goal)
Goal
```

DESCRIPTION

If `Goal` is instantiated to a structured term or atom which would be acceptable as the body of a clause, the goal `call(Goal)` is executed exactly as if that term appeared textually in place of the expression `call(Goal)`.

EXAMPLES

The following example illustrates the use of `call/1` :

```
?- [user].
Consulting user.
  jim :- printf("Hello this is Jim Rockford.\n"),
_        printf("Please leave your name and number,\n"),
_        printf("and I’ll get back to you.\n").
  user consulted.

yes.
?- call(jim).
Hello this is Jim Rockford.
Please leave your name and number,
and I’ll get back to you.

yes.
```

The following example shows how an instantiated variable can be used to run a goal:

```
?- Goal = write(Message), Message = ’Wrong Way!’,
Goal, nl.
Wrong Way!
```

```
Goal = write('Wrong Way!')
Message = 'Wrong Way!'

yes.
```

ERRORS

If `Goal` is an uninstantiated variable or a number, `call/1` will fail.

SEE ALSO

`!/0`, `:/2`,

[Clocksin 81, 6.7], [Bratko 86, 7.2], [Sterling 86, 10.4].

NAME

`catch/3`	– execute a goal, specifying an exception handler
`throw/1`	– give control to exception handler

FORMS

```
catch(Goal,Pattern,ExceptionGoal)
throw(Reason)
```

DESCRIPTION

`catch/3` provides a facility for catching errors or exceptions and handling them gracefully. Execution of `catch/3` will cause the goal Goal to be executed. If no errors or throws occur during the execution of `Goal`, then `catch/3` behaves just as if `call/1` were called with `Goal`. `Goal` may succeed thus giving control to portions of the program outside the scope of the catch. Failures in these portions of the program may cause reexecution (via backtracking) of goals in the scope of the catch of which `Goal` is the ancestor.

If a goal of the form `throw(Reason)` is encountered during the execution of `Goal` or in any subsequent reexecution of `Goal`, then the goal `ExceptionGoal` of the `catch` with the innermost scope capable of unifying `Reason` and `Pattern` together will be executed with this unification intact. Once started, the execution of the goal `ExceptionGoal` behaves just like the execution of any other goal. The goal `ExceptionGoal` is outside of the scope of the catch which initiated the execution of `ExceptionGoal` so any throws encountered during the execution of `ExceptionGoal` will be handled by catches which are ancestors of the catch which initiated the execution of `ExceptionGoal`. This means that a handler may catch some fairly general pattern, deal with some aspects for which it is prepared for, but throw back to some earlier handler for those aspects for which it is not.

Many of the builtins will cause an error if the types of the arguments to the builtin are wrong. There are other reasons for errors such as exhausting a certain resource. Whatever the cause, an error occurring during the execution of a goal causes `throw/1` to be executed. The effect is as if the goal which caused the error where replaced by a goal `throw(error(ErrorTerm, ErrorIn-`

`fo)`) where `ErrorTerm` and `ErrorInfo` supply information about the error. `ErrorTerm` is mandated by ISO Prolog Standard. `ErrorInfo` is an implementation defined (and therefore specific) term which may or may not provide additional information about the error.

The ISO Prolog Standard specifies that `ErrorTerm` be one of the following forms:

`instantiation_error` – An argument or one of its components is a variable.

`type_error(ValidType,Culprit)` – An argument or one of its components is of the incorrect type. `ValidType` may be any one of the following atoms: atom, body, callable, character, compound, constant, integer, list, number, variable. `Culprit` is the argument or component which was of the incorrect type.

`domain_error(ValidDomain,Culprit` – The base type of an argument is correct, but the value is outside the domain for which the predicate is defined. The ISO Prolog Standard states that `ValidDomain` may be any one of the following atoms: `character_code_list`, `character_list`, `close_option`, `flag_value`, `io_mode`, `not_less_than_zero`, `operator_priority`, `operator_specifier`, `prolog_flag`, `read_option`, `source_sink`, `stream_or_alias`, `stream_option`, `stream_position`, `write_option`. ALS Prolog allows `ValidDomain` to take on these additional values: `depth_computation`, `line_length`, `positive_integer`. `Culprit` is the argument which caused the error.

`existence_error(ObjectType,Culprit)` – An operation is attempted on a certain type of object specified by `ObjectType` does not exist. `Culprit` is the nonexistent object on which the operation was attempted. `ObjectType` may take on the following values: `operator`, `past_end_of_stream`, `procedure`, `static_procedure`, `source_sink`, `stream`.

`permission_error(Operation,ObjectType,Culprit)` – `Operation` is an operation not permitted on object type `ObjectType`. `Culprit` is the object on which the error occurred. `ObjectType` is an atom taking on values as described above. `Operation` may be one of the following atoms: `access_clause`, `create`, `input`, `modify`, `open`, `output`, `reposition`.

`representation_error(Flag)` – The implementation defined limit indicated by `Flag` has been breached. `Flag` may be one of the following atoms: `character`, `character_code`, `exceeded_max_arity`, `flag`.

`calculation_error(CalcFlag)` – An arithmetic operation result in an exceptional value as indicated by the atom `CalcFlag`. `CalcFlag` may take on the following values: `overflow`, `underflow`, `zero_divide`, `undefined`.

`resource_error(Resource)` – There are insufficient resources to complete execution. The type of resource exhausted is indicated by the implementation defined term `Resource`.

`syntax_error` – A sequence of characters being read by `read_term/4` can not be parsed with the current operator definitions. The reason for the syntax error (in ALS Prolog) is given in the implementation defined `ErrorInfo` (see below).

`system_error` – Other sorts of errors. These will commonly be operating system related errors such as being unable to complete a write operation due to the disk being full. Additional details about this type of error might be found in the implementation defined term `ErrorInfo`.

In ALS Prolog, `ErrorInfo` is a list providing additional information where the ISO Prolog Standard mandated term `ErrorTerm` falls short. The terms which may be on this list take the following forms:

`M:G` – The predicate in which the error occurred was in module `M` on goal `G`. Due to the compiled nature of ALS Prolog, it is not always possible to obtain all of the arguments to the goal `G`. Those which could not be obtained are indicated as such by the atom '`?`'. For this reason, the form `M:G` should be used for informational purposes only.

`errno(ErrNo)` – This form is used to further elaborate on the reason that a `system_error` occurred. The `errno/1` form indicates that a system call failed. The value of `ErrNo` is an integer which indicates the nature of the system error. The values that ErrNo take on may vary from system to system.

ALS is looking at a symbolic way of providing this information.

`syntax(Context,ErrorMessage,LineNumber,Stream)` – This form is used to provide additional information about system errors. `Context` is an atom providing some information about the context in which the error occurred. `ErrorMessage` is an atom providing the text of the message for the error. `LineNumber` is the number of the line near which the syntax error occurred. `Stream` is the stream which was being read when the syntax error occurred.

EXAMPLES

Attempt to open a non-existent file.

```
?- open(wombat,read,S).

Error: The open operation is not permitted on the
source_sink object `wombat'.
- Goal:          sio:open(wombat,read,_A,?)
- Throw pattern:
error(permission_error(open,source_sink,wombat),
                       [sio:open(wombat,read,_A,?)])
```

Define `open_for_read/2` which detects permission errors and prints a message.

```
?- consult(user).
Consulting user ...
open_for_read(File, Stream) :-
        catch( open(File, read, Stream),
           error(permission_error(open, source_sink,
File),_),
               printf(user_output,"Cannot open
%s\n",[File]) ),
           !.
user consulted
```

```
yes.
```

Try out open_for_read/2 with a non-existent file.

```
?- open_for_read(wombat,S).
Cannot open wombat

S = S

yes.
```

Try out open_for_read/2 with a variable.

```
?- open_for_read(_, S).

Error: Instantiation error.
- Goal:          sio:open(_A,read,_B,[type(text)])
- Throw pattern: error(instantiation_error,
                     [sio:open(_A,read,_B,*)])
```

Define a procedure `integer_list/2` to illustrate a use of throw/1. Note that `il/2` builds the list and throws the result back at the appropriate time.

```
?- reconsult(user).
Reconsulting user ...
integer_list(N, List) :-
       catch( il(N,[]), int_list(List), true),
       !.

il(0,L) :- throw(int_list(L)).
il(N,L) :- NN is N-1, il(NN, [N | L]).
user reconsulted

yes.
?- integer_list(8,L).

L = [1,2,3,4,5,6,7,8]

yes.
```

ERRORS

Goal is a variable

----> instantiation_error.

Goal is not a callable term

----> type_error(callable,Goal). [not yet implemented]

Reason does not unify with Pattern in any call of catch/3

----> system_error. [not yet implemented]

NOTES

In the present implementation of ALS Prolog, catch/3 leaves a choice point which is used to restore the scope of the catch when backtracked into. This choice point remains around even for determinate goals which are called from catch. Thus when catch succeeds, you should assume that a choice point has been created. If the program should be determinate, a cut should be placed immediately after the catch. It is expected that at some point in the future, this unfortunate aspect of ALS Prolog will be fixed, thus obviating the need for an explicit cut.

If throw/1 is called with Reason instantiated to a pattern which does not match Pattern in any call of catch/3, control will return to the program which started the Prolog environment. This usually means that Prolog silently exits to an operating system shell. When using the development environment, however, the Prolog shell establishes a handler for catching uncaught throws or errors thus avoiding this unceremonious exit from the Prolog system. It is occassionally possible, particularly with resource errors, to end up in this last chance handler only to have another error occur in attempting to handle the error. Since no handler exists to handle this error, control returns to the operating system often with no indication of what went wrong. .

NAME

`char_code/3` – convert between characters and codes

FORMS

```
char_codes(Char,Code)
```

DESCRIPTION

`char_code(Char,Code)` is true if the character `Char` has character code `Code`. At least one of `Char` or `Code` must be instantiated.

EXAMPLES

```
?- char_code(a,C).

C = 97

yes.
?- char_code(C,98).

C = b

yes.
?- char_code(foo,C).

Error: Argument of type character expected instead of
foo.
- Goal:          builtins:char_code(foo,_A)
- Throw pattern: error(type_error(character,foo),
                     [builtins:char_code(foo,_A)])
```

ERRORS

`Char` and `Code` are variables

——> `instantiation_error.`

`Char` is neither a variable nor a character

    ———> `type_error(character).`

`Code` is neither a variable nor an integer

    ———> `type_error(integer).`

`Code` is an integer but is not a character code

    ———> `representation_error(character_code).`

SEE ALSO

`atom_chars/2, number_chars/2, term_chars/2.`

NAME

`chdir/1` – changes the current directory to the specified directory

FORMS

```
chdir(DirName)
```

DESCRIPTION

`DirName` can be a symbol, a UIA, or a list of ASCII characters describing a valid directory. The current directory is changed to the specified directory. If the current directory cannot be changed to the given directory for any reason, this predicate fails.

EXAMPLES

```
?- chdir('../foobar').

yes.
```

NAME

`clause/2`	– retrieve a clause
`clause/3`	– retrieve a clause with a database reference
`instance/2`	– retrieve a clause from the database reference

FORMS

```
clause(Head,Body)
clause(Head,Body, Ref)
instance(Ref,Clause)
```

DESCRIPTION

When `Head` is bound to a non-variable term, the current module is searched for a clause whose head matches `Head` and whose body matches `Body`. If there is more than one clause that matches, then successive `Head`s and `Body`s will be generated upon backtracking.

When a fact is found, `Body` will be unified with the atom `true`.

`clause/3` unifies its third argument with the database reference that corresponds to the clause that was found. When `Ref` is instantiated in a call to `clause/3`, the other two arguments can be uninstantiated.

`:/2` can be used to specify which module should be searched. If `Ref` is a valid database reference, `instance(Ref, Clause)` retrieves the Prolog clause referenced by `Ref` and unifies it with `Clause`.

EXAMPLES

The following examples show the use of `clause/2 :`

```
?- listing(fruit/1).
% user:fruit/1
fruit(apple).
fruit(_34) :-
       product(_34,plantGrowth)
       ;  product(_34,plantFertilization).
fruit(orange).
```

```
% unusual:fruit/1
fruit(tomato).
fruit(kiwi).

yes.
?- clause(fruit(apple),true).

yes.

?- clause(fruit(X), Body).
X = apple
Body = true;

X = _1
Body = (product(_1,plantGrowth);
            product(_1,plantFertilization));

X = orange
Body = true;

no.
?- unusual:clause(fruit(X),Body).
X = tomato
Body = true;

X = kiwi
Body = true;

no.
```

ERRORS

If `Ref` is not instantiated to a database reference and `Head` is uninstantiated, the call to `clause/3` fails.

SEE ALSO

[Bowen 91, 7.3], [Sterling 86, 12.2], [Clocksin 81, 6.4].

NAME

`close/1`	– close an open stream
`close/2`	– close an open stream with options

FORMS

```
close(Stream_or_Alias)
close(Stream_or_Alias,CloseOptions)
```

DESCRIPTION

`close/1` and `close/2` close a stream previously opened with `open/3` or `open/4`. Closing a stream consists of flushing the buffer associated with the stream and freeing resources associated with maintaining an open stream. If there is an alias associated with the stream, this alias is disassociated and freed up for potential use by some other stream. If the stream to be closed is the current input stream, then the current input stream is set to the stream associated with the alias `user_input`. If the stream to be closed is associated with the current output stream, then the current output stream is set to the stream associated with the alias `user_output`.

Certain types of streams may have other actions performed. Atom streams opened for write will have the stream contents unified with the atom `A` which appeared in the sink specificat

ion in the call to `open`.

`Stream_or_Alias` is either a stream descriptor or an alias established via a call to `open/3` or `open/4`.

`CloseOptions` is a list consisting of options to `close/2`. The close options are:

`force(false)` – This is the default. A system error or resource error which occurs while closing the stream may prevent the stream from being closed.

`force(true)` – Errors occuring while closing the stream are ignored and resources associated with the stream are freed anyway.

ERRORS

`Stream_or_Alias` is a variable

    ——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream identifier or alias

    ——> `domain_error(stream_or_alias,Stream_or_Alias)`

`CloseOptions` is a variable

    ——> `instantiation_error.`

`CloseOptions` is neither variable nor list

    ——> `type_error(list,CloseOptions).`

`CloseOptions` is a list which contains a variable element

    ——> `instantiation_error.`

`CloseOptions` is a list which contains an element E which is neither a variable nor a valid close option

    ——> `domain_error(close_option, E).`

NOTES

Certain streams which are opened at system startup time can not be closed. Among these streams are `user_input` and `user_output`. Calling close on these aliases will neither throw an error nor really close the stream.

SEE ALSO

open/4, current_input/1, flush_output/1, *User Guide (Prolog I/O).*

NAME

`command_line/1` – provides access to start-up command line

FORMS

`command_line(SWITCHES)`

DESCRIPTION

When ALS Prolog is started from an operating system shell, the command line can be divided into system-specific and application-specific portions by use of the `-p` or `-P` (for Prolog) switch. All command line parameters to the left of the `-p` switch are treated as ALS Prolog system switches,, while those to the right of the `-p` switch are treated as application switches.

To make the latter available to Prolog applications, when ALS Prolog is initialized, a list `SWITCHES` of atoms and UIAs representing the items to the right of the `-p` switch is created, and

```
command_line(SWITCHES)
```

is asserted in module `builtins`. This assertion is always made, even when `-p` is not used, in which case the argument of `command_line/1` is the empty list.

The -P switch will force the name of the invoking program to be the argument (usually `alspro`).. the switch is useful for developing applications which will eventually be packaged (via `save_image/2`). Packaged applications will place the entire command line into `command_line/1`. In particular, the first element in the list obtained from `command_line/1` in a packaged application will be the name of the application.

EXAMPLES

```
$ alspro -p -k fast -s initstate foo
ALS-Prolog Version 2.01
   Copyright (c) 1987-94 Applied Logic Systems, Inc.

?- command_line(SW).
```

```
SW = ['-k',fast,'-s',initstate,foo]

yes.

$ alspro -P -k fast -s initstate foo
ALS-Prolog Version 2.01
   Copyright (c) 1987-94 Applied Logic Systems, Inc.

?- command_line(SW).

SW = [alspro,'-k',fast,'-s',initstate,foo]

yes.
```

NAME [Std]

`compare/3` – compares two terms in the standard order

FORMS

`compare(Relation,TermL,TermR)`

DESCRIPTION

`TermL` and `TermR` are compared according to the ***standard order*** defined below. `Relation` is unified with an atom representing the result of the comparison. `Relation` is unified with:

`=` when `TermL` is identical to `TermR`

`<` when `TermL` is before `TermR`

`>` when `TermL` is after `TermR`

The ***standard order*** provides a means to compare and sort general Prolog terms. The order is somewhat arbitrary in how it sorts terms of different types. For example, an atom is always “less than” a structure. Here’s the entire order:

Variables < Numbers < Atoms < Structured Terms

Variables are compared according to their relative locations in the Prolog data areas. Usually a recently created variable will be greater than an older variable. However, the apparent age of a variable can change without notice during a computation.

Numbers are ordered according to their signed magnitude. Integers and floating point values are ordered correctly, so `compare/3` can be used to sort numbers.

Atoms are sorted by the ASCII order of their print names. If one atom is an initial substring of another, the longer atom will appear later in the standard order.

Structured terms are ordered first by arity, then by the ASCII order of their principal functor. If two terms have the same functor and arity, then `compare/3` will recursively compare their arguments to determine the order of the two.

More precisely, if TermL and TermR are structured terms, then
TermL @< TermR holds if and only if:

the arity of TermL is less than the arity of TermR, or
TermL and TermR have the same arity, and the functor name of TermL preceeds
the functor name of TermR in the standard order, or
TermL and TermR have the same arity and functor name, and there is an integer N
less than or equal to the arity of TermL such that for all i less than N,
the ith arguments of TermL and TermR are identical, and
the Nth argument of TermL preceeds the Nth argument of TermR
in the standard order.

EXAMPLES

The following examples show the use of compare/3 :

```
?- Myself = I, compare(=, Myself, I).
Myself = _4
I = _4

yes.
?- compare(>, 100, 99).

yes.
?- compare(<, boy, big(boy)).

yes.
```

The following example shows the way structures are compared:

```
?- compare(Order,and(a,b,c),and(a,b,a,b)).
Order = ’<’
```

```
yes.
```

This says that the structure

```
and(a,b,c)
```

comes after the structure

```
and(a,b,a,b)
```

in the standard order, because the second structure has a greater arity than the first.

SEE ALSO

`==/2`, `@</2`, `sort/2`,

[Bowen 91, 7.4].

NAME

`compiletime/0` – Runs the goal only at compile time

FORMS

```
:- compiletime, Goal1, Goal2 ...
```

DESCRIPTION

`compiletime/0` is intended for use in compile-time commands within files. It will prevent any side-effects caused by

```
Goal1, Goal2, ...
```

from occurring when the *.obp* version of the file is loaded. `compiletime/0` always succeeds.

NAME

`consult/1`	– load a Prolog file
`consultq/1`	– load a Prolog file, without messages
`reconsult/1`	– load a Prolog file, updating database
`consultmessage/1`	– control printing of consulting messages
`consult_to/2`	– load a Prolog file to a module
`consultq_to/2`	– load a Prolog file to a module, without messages

FORMS

```
consult(File)
[File|Files]
consultq(File)
reconsult(File)
[-File|Files]
consultmessage(on)
consultmessage(off)
consult_to(Mod,File)
consultq_to(Mod,File)
```

DESCRIPTION

`File` should be instantiated to an atom that is the name of a file. When `consult/1` encounters a clause in a file, it compiles the clause and calls `assertz/1` to add the clause into the database. When a compile time goal or query is encountered, it is executed immediately.

`consultq/1` behaves exactly like consult, except that printing of normal messages on the terminal is suppressed. [Note, however, that if the `File` does not exist, an error message will still be printed.]

`reconsult/1` is similar to `consult/1` except in the file in question makes it possible to amend a program without having to restart from scratch and consult all the files which make up the program. Consult commands within a file which is reconsulted will act as if the specified files are textually included in the reconsulted file, and hence are reconsulted.

If the argument to `consult/1`, using either notation, is a compound term of the form `-File`, then `File` will be reconsulted.

If `File` is the atom `user`, then clauses and commands will be read in from the keyboard until an end of file.

Normally consulting messages are printed whenever a `consult` or `reconsult` command is given. If the goal

```
?- consultmessage(off).
```

is submitted, consulting messages are suppressed out until the goal

```
    ?- consultmessage(on).
```

is given. This predicate is most useful when consulting or reconsulting either under program control or from within a file.

For `consult_to/2` and `consultq_to/2`, `Mod` should be instantiated to the name of a module. Under normal usage, `File` should be instantiated to the name of a Prolog source file containing no module declarations. These two predicates behave in manners similar to their unary counterparts, except that the resulting compiled code is stored in module `Mod`.

EXAMPLES

The following example illustrates the practice of putting calls to `consult/1` inside of files to be reconsulted. First we have the three files:

```
* letters.pro
* numbers.pro
* topfile.pro
```

The following conversation with unix shows the contents of these files:

```
max:scratch$ cat letters.pro
symbol(a).
symbol(b).
symbol(c).

max:scratch$ cat numbers.pro
```

```
symbol('1').
symbol('2').
symbol('3').

max:scratch$ cat topfile.pro
symbol(x).
symbol(y).
symbol(z).
:- consult(letters).
:- consult(numbers).
```

The following conversation with unix illustrates the effects of `consult` and `reconsult`.

```
?- consult(letters).
Consulting letters ...
/max4/kev/scratch/letters consulted

yes.
?- listing.

% user:symbol/1
symbol(a).
symbol(b).
symbol(c).

yes.
?- reconsult(topfile).
Reconsulting topfile ...
Consulting letters ...
/max4/kev/scratch/letters consulted
Consulting numbers ...
/max4/kev/scratch/numbers consulted
/max4/kev/scratch/topfile reconsulted
```

```
yes.
?- listing.

% user:symbol/1
symbol(a).
symbol(b).
symbol(c).
symbol(x).
symbol(y).
symbol(z).
symbol('1').
symbol('2').
symbol('3').

yes.
```

The following interaction illustrates what happens when a couple of clauses are added to `numbers.pro` and reconsulted.

```
?- open('numbers.pro', append, _, [alias(ns)]),
?-_  writeq(ns, symbol('4')), printf(ns, '.\n', []),
?-_  writeq(ns, symbol('5')), printf(ns, '.\n', []),
?-_close(ns).

yes.
?- reconsult(numbers).
Reconsulting numbers ...
/max4/kev/scratch/numbers reconsulted

yes.
?- listing.

% user:symbol/1
symbol(a).
symbol(b).
symbol(c).
symbol(x).
symbol(y).
symbol(z).
symbol('1').
symbol('2').
symbol('3').
symbol('4').
symbol('5').

yes.
```

Note that clauses originally associated with `topfile.pro` and `letters.pro` still remain even though `numbers.pro` was reconsulted and contained clauses which defining a predicate which is also defined by `topfile.pro` and `letters.pro`.

NOTES

`reconsult/1` is not compatible with the DEC-10 notion of `reconsult`. DEC-10 reconsult would, upon seeing a procedure not already seen in a file, wipe out or abolish all clauses for that predicate before adding the new clause in question.

The present semantics of `reconsult`/1 are that all clauses which were previously defined in a file are wiped out with new clauses replacing (positionally in the procedure) any old clauses. Thus, a procedure that is defined by several files will not be entirely wiped out when `reconsult`/1 is inviked on just one of the files – only those clauses defined by the one file are wiped out and subsequently replaced.

The file *user* is special. When *user* is consulted or reconsulted, the input clauses will be taken from the user's terminal. Past versions of ALS Prolog printed a prompt when consulting *user*. The current version of ALS Prolog does not, in order that text may be easily selected and pasted in the windowed environment. The end-of-file character (often Control-D) should be used to terminate the consultation and return to the shell.

SEE ALSO

[Bratko 86, 6.5], [Clocksin 81, 6.1]

NAME

`copy_term/1` – make copy of a term

FORMS

```
copy_term(Term)
```

DESCRIPTION

`copy_term/1` will copy the term `Term` and unify this copy with `Copy`. Unbound variables in `Term` and `Copy` will not be shared between the two terms.

EXAMPLES

```
?- copy_term(f(X,g(Y,X)), Z).

X = X
Y = Y
Z = f(_A,g(_B,_A))

yes.
```

NOTES

`copy_term/1` is useful in situations involving destructive assignment. It is useful not only for the obvious situation of making a copy which is then destructively modified, but also for avoiding certain problems regarding structures becoming uninstantiated upon backtracking when using access predicates created with either `make_gv/1` or `make_hashtable/1`. See `make_gv/1` for further discussion.

SEE ALSO

`make_gv/1, make_hash_table/1, mangle/3.`

NAME

`curmod/1`	– get the current module
`modules/2`	– get the use list of a module

FORMS

```
curmod(Module)
modules(Module,Uselist)
```

DESCRIPTION

`curmod/1` instantiates `Module` to the current module.

`modules(Module,Uselist)` instantiates `Uselist` to the list of modules declared to be used by `Module`, provided `Module` is a valid module, and fails otherwise.

EXAMPLES

```
?- curmod(Module).
Module = user

yes.
?- [user].
Consulting user ...
  module foobar.
  use m1.
  use m2.
  p(a).
  endmod.
  user consulted

yes.
?- modules(foobar,X).
X = [m2,m1,builtins,user]

yes.
```

NAME

`current_input/1`	– retrieve current input stream
`current_output/1`	– retrieve current output stream

FORMS

```
current_input(Stream)
current_output(Stream)
```

DESCRIPTION

`current_input/1` will unify `Stream` with the stream descriptor associated with the current input stream. The current input stream is the stream that is read when predicates such as `get_char/1` or `read/1` are called. The current input stream may be set by calling `set_input/1`.

`current_output/1` will unify `Stream` with the stream descriptor associated with the current output stream. The current output stream is the stream which is written to when predicates such as `put_char/1` or `write/1` are called. The current output stream may be set by calling `set_output/1`.

EXAMPLES

Suppose that the file "test" is comprised of the characters "abcdefgh\n".

Open the file "test" and set the current input stream to the stream descriptor returned from open.

```
?- open(test,read,S), set_input(S).

S =
stream_descriptor('',open,file,test,[input|nooutput]
,true,

4,0,0,0,0,true,0,wt_opts(78,40000,flat),[],true,
         text,eof_code,0,0)
```

Get two characters from the current input stream.

```
?- get_char(C1), get_char(C2).

C1 = a
C2 = b
```

Close the stream associated with "test".

```
?- current_input(S), close(S).

S =
stream_descriptor('',closed,file,test,[input|nooutpu
t],true,

4,0,0,0,0,true,0,wt_opts(78,40000,flat),[],true,
         text,eof_code,0,0)
```

NOTES

`close/1` may change the current input or current output streams.

SEE ALSO

`set_input/1`, `get_char/1`, `read/1`, `put_char/1`, `write/1`, *User Guide (Prolog I/O).*

NAME

`current_op/3` – retrieve current operator definitions

FORMS

```
current_op(Priority,Specifier,Operator)
```

DESCRIPTION

`current_op/3` is used to retrieve operator definitions. Each parameter to `current_op` may be used as either an input or output argument. Logically, `current_op(Priority, Specifier,Operator)` is true if and only if `Operator` is an operator with properties defined by `Specifier` and `Priority`.

EXAMPLES

```
?- current_op(P,S,-).

P = 500
S = yfx;

P = 200
S = fy;

no.
```

NOTES

`close/1` may change the current input or current output streams.

SEE ALSO

`op/3`, `read_term/[2,3]`, *User Guide (Prolog I/O).*

NAME

`current_prolog_flag/2`– retrieve value(s) of prolog flag(s)

FORMS

`current_prolog_flag(Flag,Value)`

DESCRIPTION

`current_prolog_flag/2` is re-executable. It unifies `Flag` and `Value` to the current instantiations of the `flag/value` pairs supported by ALS Prolog.

The flags currently supported by ALS-Prolog are:

`undefined_predicate` - describes the course of action to take when an undefined predicate is called. The associated value (action) may be one of the following:

`error`	-	force an existence error when an undefined predicate is called.
`fail`	-	fail when an undefined predicate is called.
`warning`	-	warn the user when an undefined predicate is called.
`break`	-	enter the break handler when an undefined predicate is called.

ALS Prolog will eventually comply with the Draft Standard and implement the other flags defined by the standard.

ERRORS

`Flag` is a variable

——> `instantiation_error.`

`Value` is a variable

——> `instantiation_error.`

`Flag` is neither a variable nor an atom

——> `type_error(atom,Flag).`

`Value` is inappropriate for Flag
  ———> `domain_error(flag_value, Flag + Value)`

NOTES

The only flag implemented in ALS Prolog at this time is `undefined_predicate`.

SEE ALSO

set_prolog_flag/2

NAME

`dynamic/1` – declare a procedure to be dynamic

FORMS

```
dynamic(Pred/Arity)
dynamic(Module:Pred/Arity)
```

DESCRIPTION

`dynamic/1` is a procedure intended to be used in directives in source code. It will declare a procedure given by the form `Pred/Arity` or `Module:Pred/Arity` to be dynamic. Such a procedure will be considered to be defined even if it contains no clauses. Non-dynamic procedures which have no clauses are considered to be undefined and if called as such will generate a warning or error (depending on the value of the `undefined_predicate` flag). In the future, procedures declared to be dynamic will also be subject to the so called "logical database" semantics where the database will appear to be frozen once a procedure is called. Only calls that occur (temporally) after the database modification will be affected by that modification.

EXAMPLES

```
:- dynamic(foo/1).
```

NOTES

Calling `assert/1` or one of its variants for an undefined procedure will also effectively declare the procedure to be dynamic.

SEE ALSO

`consult/1, assert/[1,2].`

NAME

`edit/0`	– Invokes the default editor on last file edited
`edit/1`	– Invokes the default editor on the specified file
`vi/0`	– Invokes vi on last file edited
`vi/1`	– Invokes the vi editor on the specified file
`editorchange/1`	– Sets the default editor used by `edit/1`

FORMS

```
edit(File)
edit File
edit
vi(File)
vi File
vi
editorchange(Editor)
```

DESCRIPTION

`edit/1` Invokes the default editor on `File`. When you leave the file, it will be reconsulted. Because of the automatic `reconsult`, it's probably not a good idea to use the `edit` builtin with anything else but Prolog files. The system remembers the name of the last file that has been edited, thus allowing you to use `edit/0` when editing the same file frequently. The first time `edit/0` or `vi/0` is used, it will prompt you for the name of the file because there was no previous file. You must specify a *.pro* extension it the file has this extension. No matter what the extension, if a filename has an extension, you must enclose the full name in single quotes. A full stop (‘`.`’) should be placed after the filename because `read/1` is used for the reading. `vi/1` is equivalent to `edit/1` when the default editor is `vi`. The default editor is set by the `editorchange/1` builtin.

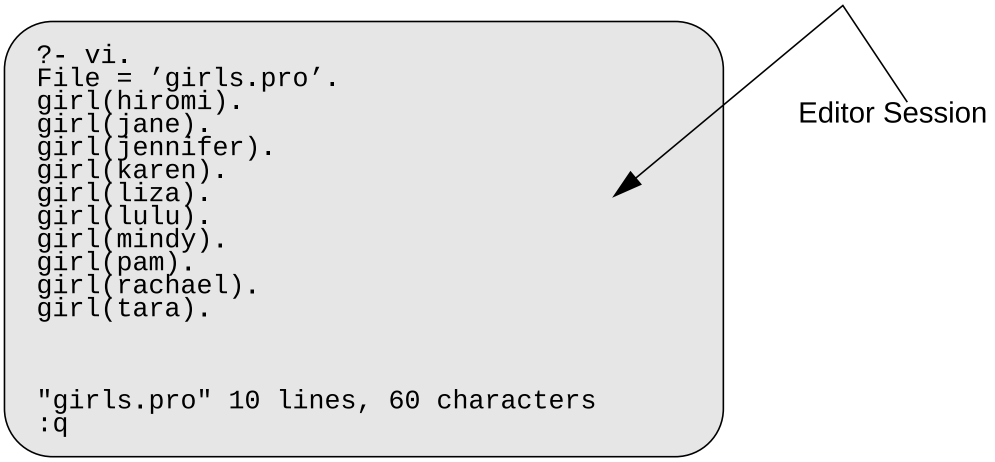
EXAMPLES

In the following example the file *girls.pro* is edited. Note that the user was prompted for the file name with the prompt `File =`. The user's typing is shown in Helvetica font.

```
?- edit.
File = 'girls.pro'.
```

The user typed in `girls.pro` followed by a full stop. The `:q` below was also typed in by the user in order to quit `vi`.

Notice that this command reconsults whether you want to or not. Be careful not to edited non-Prolog files with this command.



```
?- vi.
File = 'girls.pro'.
girl(hiromi).
girl(jane).
girl(jennifer).
girl(karen).
girl(liza).
girl(lulu).
girl(mindy).
girl(pam).
girl(rachael).
girl(tara).



"girls.pro" 10 lines, 60 characters
:q
```

```
Reconsulting girls.pro...girls.pro reconsulted.

yes.
```

SEE ALSO

`reconsult/1`, `read/1`,

[Unix Reference: `vi(1)`].

NAME

`exists/1` – tests whether a file exists

FORMS

```
exists(Filename)
```

DESCRIPTION

`Filename` is an atom representing a file name, or a path name. `exists/1` succeeds if the indicated file actually exists, and fails otherwise.

EXAMPLES

```
?-exists('foo.pro').

yes.
?-exists('../mydir/test.pro').

yes.
```

NAME

`fail/0` – always fails

FORMS

```
fail
```

DESCRIPTION

`fail/0` always fails.

EXAMPLES

```
?- fail.

no.
```

SEE ALSO

`true/0`,

[Bowen 91, 7.1], [Clocksin 81, 6.2].

NAME

’$findterm’/5 – locates the given term on the heap

FORMS

’$findterm’(Functor,Arity,HeapPos,Term,NewHeapPos)

DESCRIPTION

The term whose Functor and Arity are given is searched for on the heap starting from the given heap position HeapPos. If the term is located, the fourth argument Term is unified with the term found in the heap, and the fifth argument, NewHeapPos, is unified with a pointer to the next heap location after the term found. Heap positions are offsets with respect to the heap base. If the term cannot be found, this predicate fails.

EXAMPLES

```
?- X = f(a,b), ’$findterm’(f,2,0,T,NHP).
X = f(a,b)
T = f(a,b)
NHP = 3

yes.
```

NAME

`flush_input/1` – discard buffer contents of stream

FORMS

```
flush_input(Stream_or_Alias)
```

DESCRIPTION

`flush_input/1` will cause the buffer contents associated with the input stream `Stream_or_Alias` to be discarded. This will cause the next input operation to read in a new buffer from the source attached to `Stream`.

ERRORS

`Stream_or_Alias` is a variable

———> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

———> `domain_error(stream_or_alias,Stream_or_Alias)`

`Stream_or_Alias` is not an input stream

———>
`permission_error(input,stream,Stream_or_Alias)`

NOTES

This operation is useful on streams which have an associated output stream on which prompts are being written to. Flushing the input and then performing the requisite input operation will cause a prompt to be written out prior to reading input.

This operation is also useful for use with datagram sockets. The buffer contents associated with a datagram socket represent the entire datagram. When the end of the datagram is reached, an end-of-file condition will be triggered so that the program reading the datagram will not inadvertently read beyond the datagram into the next datagram (if any). `flush_input/1` should be used to reset the end-of-file indication after the datagram has been processed.

SEE ALSO

`open/[3,4]`, `flush_output/[0,1]`. *User Guide (Prolog I/O).*

NAME

`flush_output/0`	– flush current output stream
`flush_output/1`	– flush specific output stream

FORMS

```
flush_output
flush_output(Stream_or_Alias)
```

DESCRIPTION

A call to `flush_output/0` will cause the buffer contents associated with the current output stream to be written out.

A call to `flush_output/1` will cause the buffer contents associated with the open output stream `Stream_or_Alias` to be written out.

An output operation such as `put_char/2` or `write/2` is used to put some data out to a stream. Unless byte buffering is specified when the stream is open, the data will not be immediately output. Rather, the data will be buffered in an area associated with the open output stream. The `flush_output` built-ins cause this buffer to be written out to the sink associated with the open stream.

EXAMPLES

The following two procedures illustrate the use of `flush_output` to write out a single dot in between potentially long computations.

```
?- reconsult(user).
Reconsulting user ...
process(N) :-
        N > 0,
        put_char(user_output,'.'),
        flush_output(user_output),
        compute,
        NN is N-1,
        process(NN).
process(_) :- nl(user_output).

compute :- system('sleep 2').
user reconsulted

Yes.
?- process(5).
.....

Yes.
```

ERRORS

`Stream_or_Alias` is a variable
  ——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias
  ——> `domain_error(stream_or_alias,Stream_or_Alias)`

`Stream_or_Alias` is not associated with an open stream
  ——> `existence_error(stream,Stream_or_Alias)`

`Stream_or_Alias` is not an output stream
  ——>
  `permission_error(output,stream,Stream_or_Alias)`

`Stream_or_Alias` is a valid output stream, but the flush operation could not be completed due to some other problem detected by the operating system
  ——> `system_error`

SEE ALSO

`open/4`, `close/1`, `current_output/1`, *User Guide (Prolog I/O).*

NAME

`forcePrologInterrupt/0`– force interrupt on next call
`callWithDelayedInterrupt/1`– call goal, setting delayed interrupt
`callWithDelayedInterrupt/2`– call goal, setting delayed interrupt

FORMS

```
forcePrologInterrupt
callWithDelayedInterrupt(Call)
callWithDelayedInterrupt(Module,Call)
```

DESCRIPTION

`forcePrologInterrupt` forces an interrupt on the next call by setting the heap safety margin to a sufficiently large number which is guaranteed to be larger than the actual heap margin.

`callWithDelayedInterrupt(Call)`acts like `call/1`, invoking `Call`. However, it arranges the system so that an interrupt will take place on the first call occurring after the invocation of `Call`.

`callWithDelayedInterrupt(Module,Call)`

acts like

`callWithDelayedInterrupt/1`, except that it invokes `Call` within module `Module`.

SEE ALSO

`setPrologInterrupt/1`, `getPrologInterrupt/1`, *User Guide (Prolog Interrupts).*

NAME

`functor/3` – builds structures and retrieves information about them

FORMS

```
functor(Structure,Functor,Arity)
```

DESCRIPTION

The principal functor of term `Structure` has name `Functor` and arity `Arity`, where `Functor` is an atom. Either `Structure` must be instantiated to a term or an atom, or `Functor` and `Arity` must be instantiated to an atom and a non-negative integer respectively.

In the case where `Structure` is initially unbound, `functor/3` will unify `Structure` with a structured term of `Arity` arguments, where the principal functor of the term is `Functor`. Each argument of the new structure will be a new uninstantiated variable.

When `Structure` is instantiated to a structured term, `Functor` will be unified with the principal functor of `Structure` and `Arity` will be unified with the arity. `functor/3` treats atoms as structured terms with arity 0. The principal functor of a list is ‘`.`’ with arity 2.

EXAMPLES

```
?- functor(Structure,fish,2).
Structure = fish(_123,_124)

yes.
?- functor(city('Santa Monica', 'CA', 'USA'),
Functor, Arity).
Functor = city
Arity = 3

yes.
```

SEE ALSO

arg/3, mangle/3,

[Bowen 91, 7.6], [Clocksin 81, 6.5], [Bratko 86, 7.2], [Sterling 86, 9.2].

NAME

`gc/0` – invokes the garbage compactor

FORMS

```
gc
```

DESCRIPTION

Invokes the garbage compactor to reclaim unused space on the Prolog heap. Normally compaction is carried out automatically, so explicit calls to this predicate are not necessary.

EXAMPLES

```
?- gc.

yes.
```

SEE ALSO

[Sterling 86, 13].

NAME

`gensym/2` – generates families of unique symbolsr

FORMS

```
gensym(Prefix, Symbol)
```

DESCRIPTION

If Prefix is a symbol (either an interned or uninterned atom), then Symbol is a new UIA which has not previously existed in the system and which involves Prefix as a subsymbol. . The string also involves the system time that the current ALS Prolog image was started, together with the value of a counter for these generated symbols. Consequently, the symbols are almost guaranteed to be unique across invokations of the system, execpt for the possibility of the system clock wrapping around.

EXAMPLES

```
?- gensym(‘<Prefix>’, Symbol).

Symbol = ‘\376<Prefix>_839678026_0’

?- gensym(airplane, X).

X = ‘\376airplane_839678026_1’
```

NAME

`get_char/1`	– read a character from current input stream
`get_char/2`	– read character from a specific stream

FORMS

```
get_char(Char)
get_char(Stream_or_Alias,Char)
```

DESCRIPTION

`get_char/1` will retrieve a character from the current input stream and unify it with `Char`.

`get_char/2` will retrieve a character from the input stream associated with `Stream_or_Alias` and unify it with `Char`.

If there are no more data left in the stream to be read and if the stream has the property `eof_action(eof_code)`, then `Code` will be unified with `end_of_file`.

EXAMPLES

```
?- get_char(C1), get_char(C2), get_char(C3),
get_char(C4).
test

C1 = t
C2 = e
C3 = s
C4 = t
```

ERRORS

`Stream_or_Alias` is a variable

——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

——> domain_error(stream_or_alias, Stream_or_Alias).

Stream_or_Alias is not associated with an open stream

——> existence_error(stream,Stream_or_Alias).

Stream_or_Alias is not an input stream

——> permission_error(input,stream,Stream_or_Alias).

Char is neither a variable nor a character

——> type_error(character,Char). [See notes below]

The stream associated with Stream_or_Alias is at the end of the stream and the stream has the property eof_action(error)

——> existence_error(past_end_of_stream,Stream_or_Alias).

The stream associated with Stream_or_Alias has no input ready to be read and the stream has the property snr_action(error)

——> existence_error(stream_not_ready,Stream_or_Alias).

NOTES

A character is simply an atom with length 1. get_code/[1,2] is used to retrieve a character code.

If get_char/[1,2] is called with Char instantiated to a term which is not a character, an error will be thrown. The error thrown though will in all likelyhood be from char_code/2, not get_char/[1,2].

SEE ALSO

put_char/2, get_code/2, open/4, close/1, char_code/2, User Guide (Prolog I/O).

NAME

`get_code/1`	– read a character code from current input stream
`get_code/2`	– read character code from a specific stream

FORMS

```
get_code(Code)
get_code(Stream_or_Alias,Code)
```

DESCRIPTION

`get_code/1` will retrieve a character code from the current input stream and unify it with `Code`.

`get_code/2` will retrieve a character code from the input stream associated with `Stream_or_Alias` and unify it with `Code`.

If there are no more data left in the stream to be read and if the stream has the property `eof_action(eof_code)`, then `Code` will be unified with `-1`.

EXAMPLES

```
?- get_code(C1), get_code(C2), get_ccode(C3),
get_code(C4).
test

C1 = 116
C2 = 101
C3 = 115
C4 = 116
```

ERRORS

`Stream_or_Alias` is a variable

——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

——>

`domain_error(stream_or_alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` is not an input stream

——>
`permission_error(input,stream,Stream_or_Alias).`

`Code` is neither a variable nor a character code

——> `type_error(integer,Code).`

The stream associated with `Stream_or_Alias` is at the end of the stream and the stream has the property `eof_action(error)`

——>
`existence_error(past_end_of_stream,Stream_or_Alias)`
`.`

The stream associated with `Stream_or_Alias` has no input ready to be read and the stream has the property `snr_action(error)`

——>
`existence_error(stream_not_ready,Stream_or_Alias).`

NOTES

A character code is simply an integer restricted to a certain range of values.

SEE ALSO

`put_code/2`, `get_char/2`, `open/4`, `close/1`, `char_code/2`, User Guide (Prolog I/O),.

NAME

`get0/1`	– read the next character
`get/1`	– read the next printable character

FORMS

```
get0(Char)
get(Char)
```

DESCRIPTION

`get0/1` unifies `Char` with the ASCII code of the next character from the current input stream. If there are no more characters left in the file `Char` will be unified with `-1`.

`get/1` discards all non-printing characters from the current input stream. It unifies `Char` with the ASCII code of the first non-blank printable character. `Char` is unified with `-1` on end of file.

EXAMPLES

```
?- get(First), get(Second), get(Third).
ABCDEFGHI
First = 65,
Second = 66,
Third = 67

yes.
```

SEE ALSO

skip/1, get_char/2, get_code/2,

[Bowen 91, 7.8], [Clocksin 81, 6.9], [Bratko 86, 6.3].

NAME

`getenv/2` – gets the value of the given os environment variable

FORMS

```
getenv(EnvVar,EnvVal)
```

DESCRIPTION

`EnvVar`, which must be instantiated, can be a symbol, an UIA, or a list of ASCII characters denoting an operating system environment or shell variable. The value of this external variable is accessed and the corresponding list of ASCII characters is unified with `EnvVal`. If `EnvVar` is not defined in the os environment, `getenv/2` fails.

EXAMPLES

```
?- getenv('TERM',Term).

Term = xterm

yes.
?- getenv('PATH',Path).

Path = '.:/usr/local/bin:/usr/bin/X11:/usr/bin:/
bin:/usr/ccs/bin'

yes.
?- getenv('FOOBAR',Foobar).

no.
```

NAME

`halt/0` – exit ALS Prolog

FORMS

```
halt
```

DESCRIPTION

`halt/0` causes the ALS Prolog system to exit, returning control to the operating system shell. It can either be invoked from the top level of the ALS Prolog shell, or from within a running program.

EXAMPLES

In this example, `halt/0` is called from the Prolog shell on a Unix C shell system on a machine named ’wizard’:

```
?- halt.
wizard%
```

NOTES

On most systems, typing the end of file characters after the `?-` prompt of the Prolog shell will also cause ALS Prolog to exit to the operating system shell. On Unix, the end of file character is entered by typing `Control-D.`On DOS and Win32, the end of file character is `Control-Z`. On the Mac, either `Control-D` or `Control-Z` can be used as the end of file character; in addition, the Quit menu can also be used.

SEE ALSO

`abort/0`.

BUGS

`halt/0` does not close any streams nor does it flush their output buffers.

NAME

`is/2` – evaluates an arithmetic expression

FORMS

```
Result is Expression
```

DESCRIPTION

`Expression` should be a ground term that can be evaluated. Numbers evaluate as themselves, and a list evaluates as the first element of the list. The operators listed in Table 9 (*Arithmetic Operators.*) and Table 11 (*Arithmetic Functions.*) can also be evaluated when their arguments can be evaluated. If `Result` is an unbound variable, then it will be bound to the numeric value of `Expression`. If `Result` is not unbound, then it will be evaluated, and the value of the `Result` will be unified with the value of the `Expression.`

Table 8:

Operator	Description
`-X`	unary minus
`X div Y`	integer division
`X mod Y`	`X` (integer) modulo `Y`
`X xor Y`	`X` exclusive or `Y`
`X*Y`	multiplication
`X+Y`	addition
`X-Y`	subtraction
`X/Y`	division
`X//Y`	integer division

Table 8:

Operator	Description
`X/\Y`	integer bitwise conjunction
`X<<Y`	integer bitwise left shift of `X` by `Y` places
`X>>Y`	integer bitwise right shift of `X` by `Y` places
`X\/Y`	integer bitwise disjunction
`X^Y`	`X` to the power `Y`
`\X not(X)`	integer bitwise negation
`0'Char`	the ASCII code of `Char`

Table 9. Arithmetic Operators.

Table 10:

Function	Description
`abs(X)`	absolute value
`acos(X)`	arc cosine
`asin(X)`	arc sine
`atan(X)`	arc tangent
`cos(X)`	cosine
`cputime`	CPU time in seconds since ALS Prolog started.
`exp(X)`	natural exponential function
`exp10(X)`	base `10` exponential function
`floor(X)`	the largest integer not greater than `X`

Table 10:

Function	Description
`heapused`	heap space in use, in bytes
`j0(X)`	Bessel function of order `0`
`j1(X)`	Bessel function of order `1`
`log(X)`	natural logarithm
`log10(X)`	base `10` logarithm
`random`	returns a random real number between `0` and `1`
`realtime`	actual time in seconds since ALS Prolog started.
`round(X)`	integer rounding of `X`
`sin(X)`	sine
`sqrt(X)`	square root
`tan(X)`	tangent
`trunc(X)`	the largest integer not greater than `X`
`y0(X)`	Bessel function of second kind of order `0`
`y1(X)`	Bessel function of second kind of order `1`

Table 11. Arithmetic Functions.

EXAMPLES

```
?- 2 is 3-1.

yes.

?- X is 6*7.
X = 42
```

```
yes.

?- X is 2.5 + 3.5.
X = 6

yes.
```

ERRORS

`is/2` fails when it attempts to evaluate an unknown operator, or if `Expression` is not ground. Failure also occurs if there are any arithmetic faults, such as overflow, underflow, or division by zero.

NOTES

ALS Prolog plans to comply to the ISO Prolog Standard regarding errors. A calculation error will be thrown on overflow, underflow, division by zero, or use of an unrecognized arithmetic operator.

SEE ALSO

[Bowen 91, 7.7], [Clocksin 81, 6.11], [Bratko 86, 3.4].

NAME

`leash/1` – set which ports are leashed for the debugger

FORMS

```
leash(Mode)
leash([Mode|Modes])
```

DESCRIPTION

The leashing mode of the debugger is set to `Mode` where `Mode` is one of the atoms shown in Table 13 (*Leashing Modes.*) . Note that it does not make sense to use the list format for specifying modes if the `all` mode is included with other modes.

Table 12:

Mode	Function
`all`	Prompt at `all` ports
`call`	Prompt at `call` ports
`exit`	Prompt at `redo` ports
`fail`	Prompt at `fail` ports
`redo`	Prompt at `redo` ports

Table 13. Leashing Modes.

EXAMPLES

The following examples illustrate the use of `leash/1`:

```
?- leash([call,redo]).

yes.
```

```
?- leash([]).

yes.
```

Note that using an empty list as the argument to `leash/1`, as shown in the example above, results in no ports being leashed.

SEE ALSO

`trace/1`, `spy/1`, *Tools (Using the Debugger), [Clocksin 81, 8.4]*

NAME

`length/2` – count the number of elements in a list

FORMS

```
length(List,Size)
```

DESCRIPTION

`Size` is unified with the number of elements in `List`.

EXAMPLES

```
?- length([a,b,c],X).
X = 3

yes.
```

NOTES

`length/2` is defined by:

```
length(List,Length) :- length(List,0,Length).
length([],Length,Length) :- !.
length([_|Rest],Old,Length) :-
        New is Old+1,
        length(Rest,New,Length).
```

NAME

`listing/0`	– Prints all clauses
`listing/1`	– Prints clauses matching the specified template

FORMS

```
listing
listing(Pred)
listing(Pred/Arity)
listing(Mod:Pred/Arity)
```

DESCRIPTION

`listing/0` lists all the clauses in the database except those in the `built-ins` module, and several other system modules.

If `Pred` is the name of a predicate, `listing(Pred)` causes all the clauses for `Pred` of any arity and residing in any module other than `builtins` to be listed to the current output stream. The argument `Pred` may be a predicate specification of the form `Name/Arity` in which case only the clauses for the specified predicate are listed. If `P` is the name of predicate of arity `A` which is defined in module `M`, then

```
    ?- listing(M:P/A).
```

will cause only the clauses of the definition of `P/A` in `M` to be listed to the current output stream. A form of wildcards can be used if some of the arguments are left as uninstantiated variables. The following example lists all the `eggs` clauses in module `dairy` regardless of what their arity is:

```
    ?- listing(dairy:eggs/_).
```

To list all of the clauses in the module `dairy`, you could submit the goal:

```
    ?- listing(dairy:_).
```

SEE ALSO

[Clocksin 81, 6.4].

NAME

`make_gv/1`	– create named global variable and access method
`make_det_gv/1`	– create named global variable and access methods which preserves instantiations of structures
`free_gv/1`	– release store associated with named global variable

FORMS

```
make_gv(Name)
make_det_gv(Name)
free_gv(Name)
```

DESCRIPTION

`make_gv/1` allocates an internal global variable and creates two access predicates called `setNAME/1` and `getNAME/1` where `NAME` is the atom `Name`. These access predicates are installed in the module from which `make_gv/1` is called.

The `setNAME/1` predicate is used to set the allocated global variable to the term given to `setNAME` as its only argument. This operation is safe in that the contents of the global variable will survive backtracking without any dangling references. Care should be taken when using these global variables with backtracking as it is easy to create a ground structure in which "holes" will appear upon backtracking. These holes are uninstantiated variables where there used to be a term. They are caused by some bit of non-determinism when creating the term. If the non-determinism is removed via cut prior to a global variable operation, these "holes" will often not show up upon backtracking. If the non-determinism is removed after the global variable operation takes place, these holes will very likely show up. The reason that this is so is because the global variable mechanism will (as a consequence of making the structure safe to backtrack over) eliminate the ability of cut to discriminate among those trail entries which may be safely cut and those which are needed in the event of failure.

In situations where this is a problem, a call to `copy_term/2` may be used to create a copy of the term prior to setting the global variable. The instantiation of the term that exists at the time of the copy will be the instantiation of the term which survives backtracking over the copy operation.

Also of interest is the time complexity of the set operation. So long as the argument to `setNAME/1` is a non-pointer type, that is a suitably small integer or certain types of atoms (the non-UIA variety), the set operation is a constant time operation. Otherwise it requires time linearly proportional to the current depth of the choice point stack.

The `getNAME/1` predicate created by `make_gv/1` is used to get the contents of one of these global variables. The contents of the global variable is unified with the single parameter passed to getNAME/1.

`make_det_gv/1` creates access methods just like `make_gv/1` but the `setNAME/1` method avoids the problems referred to above concerning certain instantiations in structure becoming undone. It does this by making a copy of the term prior to setting the global variable. Making a copy of the term has the disadvantage of the increased space and time requirements associated with making copies.

`free_gv/1` removes access methods created by `make_gv/1` and frees up the global variable.

EXAMPLES

```
--------
%% gvdemo1 – demonstrate the subtleties of combining
global
%variables with backtracking

:- make_gv('_demo').%% Create get_demo/1 and
set_demo/1.

print_demo(N) :- get_demo(X), printf('demo%d:
%t\n',[N,X]).

demo1 :- demo1(_).
demo1 :- print_demo(1).
demo1(_) :- X=f(Y), (Y=i ; Y=j), set_demo(X),
print_demo(1), fail.

demo2 :- demo2(_).
demo2 :- print_demo(2).
demo2(_) :- X=f(Y), (Y=i ; Y=j),!,set_demo(X),
print_demo(2), fail.

demo3 :- demo3(_).
demo3 :- print_demo(3).
demo3(_) :- X=f(Y), (Y=i ; Y=j),
set_demo(X),!,print_demo(3), fail.

demo4 :- demo4(_).
demo4 :- print_demo(4).
demo4(Y) :- X=f(Y), (Y=i ; Y=j),!,set_demo(X),
print_demo(4), fail.

demo5 :- _=f(Y), set_demo([a]), demo5(Y).
demo5 :- print_demo(5).
demo5(Y) :- X=f(Y), (Y=i ; Y=j),!,set_demo(X),
print_demo(5), fail.

demo6 :- set_demo([a]), _=f(Y), demo6(Y).
demo6 :- print_demo(6).
demo6(Y) :- X=f(Y), (Y=i ; Y=j),!,set_demo(X),
print_demo(6), fail.

demo7 :- demo7(_).
demo7 :- print_demo(7).
demo7(_) :- X=f(Y), (Y=i ; Y=j), copy_term(X,Z),
            set_demo(Z), !, print_demo(7), fail.

demo :- demo1, nl, demo2, nl, demo3, nl,
           demo4, nl, demo5, nl, demo6, nl,
           demo7.
```

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

The above program demonstrates the subtelties of combining global variables with backtracking. Here is a sample run of this program:

```
?- demo.
demo1: f(i)
demo1: f(j)
demo1: f(_A)

demo2: f(i)
demo2: f(i)

demo3: f(i)
demo3: f(_A)

demo4: f(i)
demo4: f(i)

demo5: f(i)
demo5: f(_A)

demo6: f(i)
demo6: f(i)

demo7: f(i)
demo7: f(i)
```

In each of these seven different tests, some non-determinism is introduced through the use of `;/2`.

`demo1` makes no attempt eliminate this non-determinism. Yet the results might be somewhat surprising. `set_demo/1` is called twice; once with `X` instantiated to `f(i)`, the second time with `X` instantiatedto `f(j)`. Yet when we fail out of `demo1/1`, `print_demo/1` reports the "demo" variable to have an uninstantiated portion.

`demo2` eliminates the non-determinism in a straightforward fashion through the use of a cut. Here the `f(i)` is made to "stick".

`demo3` is a slight variation on `demo2`. It shows that eliminating determinism after setting the global variable is too late to make the instantiations "stick".

`demo4` is similar to demo3, but shows that it is alright for `Y` to be "older" than the structure containing it.

`demo5` shows that an intervening global variable operation may screw things up by making `Y` live in a portion of the heap which must be trailed when `Y` is bound. The cut prior to setting the global variable is not permitted to remove the trail entry which eventually causes `Y` to lose its instantiation.

`demo6` shows that creating the variable after the global variable operation has the same effect as `demo4`.

`demo7` demonstrates a technique that may be used to always make instantiations "stick". It creates a new copy of the term and calls `set_demo/1` with this new copy.

If the call to `make_gv/1` at the top of the file were replaced with a call to `make_det_gv/1`, then all of the instantiations would "stick" as `make_det_gv/1` automatically makes a copy of the term thus doing implicitly what `demo7` does explicitly.

BUGS

`free_gv/1` does not work for access methods created by `make_det_gv/1`.

SEE ALSO

`make_hash_table/1, copy_term/2, mangle/3.`

NAME

`make_hash_table/1` – create hash table and access predicates

FORMS

`make_hash_table(Name)`

DESCRIPTION

`make_hash_table/1` will create a hash table and a set of access methods with the atom `Name` as the suffix. Suppose for the sake of the following discussion that `Name` is bound to the atom '`_table`'. Then the access predicates created will be as follows:

`reset_table` – throw away old hash table associated with the '`_table`' hash table and create a brand new one.

`set_table(Key,Value)` – associate `Key` with `Value` in the hash table `Key` should be bound to a ground term. Any former associations that `Key` had in the hash table are replaced.

`get_table(Key,Value)` – get the value associated with the ground term bound to `Key` and unify it with `Value.`

`del_table(Key,Value)` – delete the `Key/Value` association from the hash table. `Key` must be bound to a ground term. `Value` will be unified against the associated value in the table. If the unification is not successful, the table will not be modified.

`pget_table(KeyPattern,ValPattern)` – The "`p`" in `pget` and `pdel`, below, stands for pattern. `pget_table` permits `KeyPattern` and `ValPattern` to have any desired instantiation. It will backtrack through the table and locate associations matching the "pattern" as specified by `KeyPattern` and `ValPattern`.

`pdel_table(KeyPattern,ValPattern)` – This functions the same as `pget_table` except that the association is deleted from the table once it is retrieved.

EXAMPLES

```
?- make_hash_table('_assoc').

yes.

?- set_assoc(a, f(1)).

yes.
?- set_assoc(b, f(2)).

yes.
?- set_assoc(c, f(3)).

yes.
?- get_assoc(X, Y).

no.
?- get_assoc(c, Y).

Y = f(3)

yes.
?- pget_assoc(X, Y).

X = c
Y = f(3);

X = b
Y = f(2);

X = a
Y = f(1);

no.
?- del_assoc(b, Y).
```

```
Y = f(2)

yes.
?- pdel_assoc(X, f(3)).

X = c

yes.
?- pget_assoc(X, Y).

X = a
Y = f(1);

no.
?- reset_assoc.

yes.
?- pget_assoc(X,Y).

no.
```

NOTES

Unlike `assert` and `retract`, the methods created by `make_hash_table/1` do not access the database. The associations between keys and values is stored on the heap. Thus elements of either keys or values may be modified in a destructive fashion. This will probably not have desirable consequences if a key is modified.

These predicates have an advantage over `assert` and `retract` in that no copies are made. In fact structure may be shared between hash table entries.

See the discussion in `make_gv/1` concerning global variable modification and backtracking.

SEE ALSO

`make_gv/1`.

NAME

`mangle/3` – destructively modify a structure

FORMS

```
mangle(Nth,Structure,NewArg)
```

DESCRIPTION

mangle/3 destructively modifies an argument of a compound term in a spirit similar to Lisp's `rplaca` and `rplacd`. `Structure` must be instantiated to a compound term with at least `N` arguments. The `Nth` argument of `Structure` will become `NewArg`. Lists are considered to be structures of arity two.

Modifications made to a structure by `mangle/3` will survive failure and backtracking.

Even though `mangle/3` implements destructive assignment in Prolog, it is not necessarily more efficient than copying a term. This is due to the extensive cleanup operation which ensures that the effects of a `mangle/3` persist across failure.

EXAMPLES

```
?- Victim = doNot(fold,staple,mutilate),
mangle(2,Victim,spindle).
Victim = doNot(fold,spindle,mutilate)
yes.
```

SEE ALSO

`arg/3`.

NAME

`member/2`	– list membership
`dmember/2`	– list membership

FORMS

```
member(Element,List)
dmember(Element,List)
```

DESCRIPTION

`member/2` succeeds when `Element` can be unified with one of the elements in the list, `List`. `dmember/2` is the determinate version of `member/2`.

EXAMPLES

```
?- member(a,[a,b,c]).

yes.
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
no.
```

NOTES

`member/2` and `dmember/2` are defined by the following clauses:

```
member(Item,[Item|_]).
member(Item,[_|Rest]) :- member(Item,Rest).

dmember(Item,[Item|_]) :- !.
dmember(Item,[_|Rest]) :- dmember(Item,Rest).
```

NAME

`module_closure/2`	– creates a module closure
`module_closure/3`	– creates a module closure for the specified procedure

FORMS

```
:- module_closure(Name,Arity,Procedure).
:- module_closure(Name,Arity).
```

DESCRIPTION

For some Prolog procedures, it is essential to know the module within which they are invoked. For example, `setof/3` must invoke the goal in its second argument relative to the correct module. The problem is that `setof/3` is defined in module `builtins`, while it may invoked in some other module which is where the code defining the goal in the second argument should be run. In reality, `setof/3` is defined as the module closure of another predicate `setof/4` (whose definition appears in the `builtins` module). The extra argument to `setof/4` is the module in which the goal in the second argument of `setof/3` is to be run. Declaring `setof/3` to be a module closure of `setof/4` means that goals of the form

```
...,setof(X, G, L),...
```

are automatically expanded to goals of the form

```
...,setof(M, X, G, L),...
```

where `M` is the current module; i.e., the module in which the original call took place. Thus `setof/4` is supplied with the correct module `M` in which to run the goal in the second argument of the original call to `setof/3`.

The actual predicate that you write should expect to receive the calling module as its first argument. Then one ‘closes’ the predicate with a module closure declaration which suppresses the first (module) argument. The arguments to `module_closure/3` are as follows:

- `Name` is the name of the procedure the user will call.

- `Arity` is the number of arguments of the user procedure; that is, the number of arguments in the ‘closed’ procedure which the user procedure will call.
- `Procedure` is the name of the (unclosed) procedure to call with the additional module argument. Note that `Procedure` can be different than `Name`, although they are often the same.

The procedure that the user will call should be exported if it is contained within a module. The actual (unclosed) procedure does not need to be exported. `module_closure/2` simply identifies the first and third arguments of `module_closure/3`. That is, the command

```
:- module_closure(foo,5).
```

is equivalent to

```
       :- module_closure(foo,5,foo).
```

EXAMPLES

The following example illustrates the use of `module_closure/3`. First assume that the following three modules have been created and loaded:

```
module m1.
use m3.

export testA/1.
testA(X) :- leading(X).
p(tom).
p(dick).
p(harry).
endmod. % m1

module m2.
use m3.

export testB/1.
testB(X) :- leading(X).
p(sally).
```

```
p(jane).
p(martha).
endmod. % m2

module m3.
leading(X) :- p(X).
endmod. % m3
```

Attempting to run either `testA` or `testB` fails:

```
?- testA(X).

no.
?- testB(X).

no.
```

This is because the call to `p(X)` runs in module `m3` which has no clauses defining `p/1`. Now let us change module `m3` to read as follows:

```
module m3.
first(M,X) :-  M:p(X).

export leading/1.
:- module_closure(leading,1,first).
endmod.
```

We have defined a new predicate `first/2` which carries a module as its first argument and which makes the call to `p(X)` in that module. And we have specified that `leading/1` is the module closure of `first/2`. Now the calls succeed:

```
?- testA(X).
X = tom
yes.
?- testB(X).
X = sally
yes.
```

Note that we exported `leading/1` from module `m3`, and both module `m1` and module `m2` were declared to use module `m3`.

SEE ALSO

`:/2,` *User Guide (Modules)*.

NAME

`name/2` – converts strings to atoms and atoms to strings

FORMS

```
name(Constant,PrintName)
```

DESCRIPTION

When `Constant` is instantiated to an atom or a number, `PrintName` is unified with a list of ASCII codes that correspond to the printed representation of `Constant`. When `PrintName` is a list of ASCII codes, `Constant` will be unified with the atom or number whose printed representation is the string `PrintName`.

```
?- name(Symbol, [0’a,0’l,0’i,0’e,0’n,0’s]).
Symbol = aliens
yes.
?- name(aliens, "aliens").

yes.
?- name([], "[]").

yes.
?- name(2018, X).
X = [50,48,49,56]
yes.
?- name(X, "2018").
X = 2018          % 2018 is an integer, not a symbol
yes.
```

NOTES

We recommend the use of `atom_chars/2` and `number_chars/2` over `name/2`.

SEE ALSO

`atom_chars/2`, `atom_codes/2`, `number_chars/2`, `number_codes/2`, `term_chars/2`, `term_codes/2`, *User Guide (Syntax of ALS Prolog),, [Bowen 91, 7.8], [Clocksin 81, 6.5], [Bratko 86, 6.4], [Sterling 86, 12.1].*

NAME

`nl/0`	– write a newline character to the current output stream
nl/1	– write a newline character to a specified stream

FORMS

```
nl
nl(Stream_or_Alias)
```

DESCRIPTION

`nl/0` writes out a newline character (sequence) to the current output stream.

`nl/1` writes out a newline character (sequence) to the output stream associated with `Stream_or_Alias`.

The character sequence written depends on the `write_eoln_type` option used when opening the stream. The defaults for the `write_eoln_type` are `cr(“\r”)` on MacOS, `lf(“\n”)` on unix, and `crlf(“\r\n”)` on MS DOS and Windows.

EXAMPLES

```
?- write(start), nl, write(finish).
start
finish

yes.
```

ERRORS

Stream_or_Alias is a variable

`——>  instantiation_error`

`Stream_or_Alias` is a variable is neither a variable nor a stream descriptor nor an alias

`——>  domain_error(stream_or_alias,Stream_or_Alias)`

`Stream_or_Alias` is not associated with an open stream

```
    ——>  existence_error(stream, Stream_or_Alias)
```

`Stream_or_Alias` is not an output stream

```
    ——>
    permission_error(output,stream,Stream_or_Alias)
```

SEE ALSO

`put_char/1`, `put_code/1`, User Guide (Prolog I/O), [Bowen 91, 7.8], [Bratko 86, 6.2.1], [Clocksin 81, 6.9].

NAME

`not/1`	– tests whether a goal fails
`\+/1`	– tests whether a goal fails

FORMS

```
not Goal
not(Goal)
\+ Goal
\+(Goal)
```

DESCRIPTION

`not/1` and `\+/1` implement negation by failure. If the `Goal` fails, then `not(Goal)` succeeds. If `Goal` succeeds, then `not(Goal)` fails. When `not/1` succeeds it doesn't bind any variables. Cuts occurring within `Goal` will be restricted to cutting choices created within the execution of `Goal`.

EXAMPLES

```
?- not(true).

no.
?- not(a=b).

yes.
?- not(not(f(A)=f(b))).
A = _2
yes.
```

SEE ALSO

[Bowen 91, 7.1], [Sterling 86, 11.3], [Bratko 86, 5.3], [Clocksin 81, 6.7].

NAME

`number_chars/2`	– convert between a number and the list of characters which represent the number
`number_codes/2`	– convert between a number and the list of character codes which represent the number

FORMS

```
number_chars(Number,CharList)
number_codes(Number,CodeList)
```

DESCRIPTION

If `CharList` is bound to a list of characters then it is parsed according to the syntax rules for numbers. Should the parse be successful, the resulting value is unified with `Number` in a call to `number_chars/2`.

If `CodeList` is bound to a list of character codes then it is is parsed according to the syntax rules for numbers. Should the parse be successful, the resulting value is unified with `Number` in a call to `number_codes/2`.

In `Number` is bound to a number in either `number_chars/2` (or `number_codes/2`), after first ascertaining that `CharList` (or `CodeList`) is bound to a ground list, then `CharList` (or `CodeList`) will be bound to a list of characters (character codes) that would result as output from `write_canonical(Number)`.

EXAMPLES

```
?- number_chars(-2.3,L).

L = [-,'2',.,'3']

yes.
?- number_codes(N,"123").

N = 123
```

```
yes.
?- number_codes(N,"   123.40000000000000").

N = 123.4

yes.
?- number_chars(123.4,['1',A,B,.,C]).

A = '2'
B = '3'
C = '4'

yes.
?- number_codes(N,"0xffe").

N = 4094

yes.
?- number_codes(N,"foobar").

Error: Syntax error.
- Goal:          builtins:number_codes(_A,"foobar")
- Throw pattern:
error(syntax_error,[builtins:number_codes(_A,*)])
```

ERRORS

`Number` and `CharList` are variables (`number_chars/3`)

———> `instantiation_error.`

`Number` and `CodeList` are variables (number_codes/3)

———> `instantiation_error.`

`Number` is neither a number nor a variable

———> `type_error(number,Number)`

`CharList` is neither a variable nor a list of characters

——> `domain_error(character_list,List)`

`CodeList` is neither a variable nor a list of character codes

——> `domain_error(character_code_list,List)`

`CharList` (or `CodeList`) is not parsable as a number

——> `syntax_error`

SEE ALSO

`read_term/3`, `write_canonical/2`.

NAME

`op/3` – define operator associativity and precedence

FORMS

```
op(Precedence,Associativity,Atom)
op(Precedence,Associativity,[Atom|Atoms])
```

DESCRIPTION

If `Precedence` is instantiated to an integer between `0` and `1200` then `Associativity` must be one of the indicators found in Table 15 (*Prolog Operator Types.*) . `Atom` can be any atom, but should not require the use of single quotes (`'`) in order to be parsed.

Table 14:

Indicator	Interpretation
`xfy`	infix -- right associative
`yfx`	infix -- left associative
`xfx`	infix -- non-multiple
`fy`	prefix -- multiple
`fx`	prefix -- non-multiple
`yf`	postfix -- multiple
`xf`	postfix -- non-multiple

Table 15. Prolog Operator Types.

If `Precedence` is uninstantiated, but `Associativity` and `Atom` correspond to an existing operator, `Precedence` will be unified with the previously declared precedence of the operator. The default operator precedences and associativities can be found in Table 17 (*Default Operator Associativity and Precedence.*) .

A number of convenience predicates are declared as prefix operators (`fx`) of precedence 1125. These include:

- `trace`, `module`, `use`, `export`, `dynamic`
- `cd dir ls edit vi`

Table 16:

Operator(s)	Associativity	Precedence
`:-`	`fx` and `xfx`	1200
`?-`	`fx`	1200
`-->`	`xfx`	1200
`;`	`xfy`	1100
`->`	`yfx`	1050
`,`	`xfy`	1000
`:`	`yfx`	950
`+  not`	`fy`	900
`.`	`xfy`	800
`spy nospy`	`fx`	800
`=   \=`	`xfx`	700
`==  \==`	`xfx`	700
`@<   @> @=<   @>=`	`xfx`	700
`is`	`xfx`	700
`=:=   =\=`	`xfx`	700

Table 16:

Operator(s)	Associativity	Precedence
`<  >  =<  >=`	`xfx`	`700`
`=..`	`xfx`	`700`
`+`	`yfx`	`500`
`-`	`yfx`	`500`
`\/  /\`	`yfx`	`500`
`*  //  /  div  >>  <<`	`yfx`	`400`
`mod`	`yfx`	`300`
`+  -  \`	`fy`	200
`^`	`yfx`	`200`

Table 17. Default Operator Associativity and Precedence.

EXAMPLES

```
?- op(200,yfx,to), op(200,yfx,on).

yes.
?- display(get to work on time).
on(to(get,work),time)
yes.
```

SEE ALSO

[Bowen 91, 7.9], [Sterling 86, 8.1], [Bratko 86, 3.3], [Clocksin 81, 5.5].

NAME

`open/3`	– open a stream
`open/4`	– open a stream with options

FORMS

```
open(Name,Mode,Stream)
open(Name,Mode,Stream,Options)
```

DESCRIPTION

`open/3` and `open/4` are used to open a file or other entities for input or output. Calling `open/3` is the same as calling `open/4` where `Options` is the empty list.

Once a stream has been opened with `open/3` or `open/4`, `read_term/3` or `write_term/3` might then be used to read or write terms to the opened stream. `get_char/2` or `put_char/2` can be used to read or write characters. A ***source*** refers to some entity which may be opened as a stream for read access. Such streams, once open, are called input streams. A ***sink*** refers to some entity which may be written to. Such a stream is called an output stream. There are streams which are both sources and sinks; such streams may be both read from and written to.

`Name` specifies the source/sink to be opened. Whether `Name` is a source or a sink will depend on the value of `Mode`. `Mode` may either be `read`, indicating that Name is a source or `write`, which indicating that `Name` is a sink. If the source/sink specified by `Name`, `Mode`, and the options in `Options` is successfully opened, then `Stream` will be unified with a stream descriptor which may be used in I/O operations on the stream. If the stream could not be successfully opened, then an error is thrown.

If `Name` is an atom, then the contents of the stream are the contents of the file with name `Name`. `Mode` may take on additional values for file streams. The values which `Mode` may take on for file streams are:

`read`	– open the file for read access
`write`	– open the file for write access; truncate or create as

	necessary
`read_write`	– both reading and writing are permitted; file is not truncated on open
`append`	– open file with write access and position at end of stream

If `Name` has the form `atom(A)`, then `Name` represents an atom stream. When opened with `Mode` equal to `read`, `A` must be an atom. The contents of the stream are simply the characters comprising the atom `A`. When opened for write access, `A` will be unified with the atom formed from the characters written to the stream during the time that the stream was open. The unification is carried out at the time that the stream is closed.

If `Name` has the form `code_list(CL)` or `string(CL)`, then `Name` represents a (character) code list stream. When opened for read access, the contents of such a stream are the character codes found in the list `CL`. When opened for write access, `CL` will be unified with a list of character codes. This list is formed from the charcters written to the stream during the lifetime of the stream. The unification is carried out when the stream is closed.

If `Name` has the form `char_list(CL)`, then `Name` represents a character list stream. The behavior of character list streams is identical to that of code list streams with the exception that the types of the objects in the lists are different. A code list consists of character codes, whereas a character list consists of a list of characters.

`Name` may also take on one of the following forms representing a socket:

```
    socket(unix,PathName)
    socket(inet_stream,Host)
    socket(inet_stream,Host,Port)
    socket(inet_dgram,Host,Port)
    socket(clone,Stream_Or_Alias)
```

Regardless of the mode (`read` or `write`), the call to `open/[3,4]` will attempt to open the stream as a server (if appropriate). Failing that, it will attempt to open the stream as a client. If a connection-oriented socket is opened as a server, then prior to the first buffer `read` or `write`, the stream will wait for a client to connect. If the stream is opened as a client, the connection (in a connection-oriented socket) is established as part of the `open`.

`socket(unix,PathName)`: Open a unix domain socket. Addresses in the unix domain are merely path names which are specified by `PathName`. Unix domain sockets are somewhat limited in that both the server and client process must reside on the same machine. The stream will be opened as a server if the name specified by `PathName` does not currently exist (and the requisite permissions exist to create a directory entry). Otherwise, the stream will be opened as a client.

`socket(inet_stream,Host)`: Open an internet domain stream socket on the host given by `Host` using port number 1599. The stream will be opened as a server if `Host` is set to the name of the host on which the process is running and no other process has already established a server stream on this port. Otherwise, the stream will be opened as a client. A permission error will be generated if neither operation can be performed. The host specification may either be a host name or an internet address.

`socket(inet_stream,Host,Port)`: Similar to the above, but the `Port` may specified. This permits an application to choose its own "well known" port number and act as either a server or client. Alternately, both `Host` and `Port` may be variable in which case the system will open a stream at a port of its choosing. When variable, `Host` and `Port` will be instantiated to values of the current host and the port which was actually opened.

`socket(inet_dgram,Host,Port)`: Similar to `inet_stream`, but a datagram socket is created instead. A datagram socket is an endpoint which is not connected. The datagram socket will be opened as a server socket if the Host is either variable or bound to the name of the current host and `Port` is either variable or bound to a port number which is not currently in use. Otherwise, a client socket is established which (by default) will write to the host and port indicated by `Host` and `Port`. A server socket is initially set up to write out to UDP port 9 which will discard any messages sent. Datagram sockets will set the end-of-file indication for each datagram read. This mechanism permits code written in Prolog to fully process each incoming datagram without having to worry about running over into the next datagram. `flush_input/1` should be used to reset the stream attached to the datagram in order for more input to be read. Programmers using datagrams should strive to make each datagram self contained. If this is a hardship, stream sockets should be used instead.

`socket(clone,Stream_or_Alias)`: This specification will create a new (prolog) stream descriptor for a socket from an existing socket stream descriptor. This gives the programmer the ability to create more than one buffer which refers to the same socket. This cloning mechanism has two uses. Firstly, sockets are full duplex which means that they may be both read from and written to. Yet, the interface which ALS Prolog provides will only naturally provide read access or write access, not both simultaneously. The cloning mechanism accommodates this problem by allowing separate (prolog) stream descriptors for each mode which refer to the same unix socket descriptor. Secondly, server socket streams will only act as a server until a connection is established. Once the connection is made, they lose their "server" property. An application which wants to service more than one client will want to clone its "server" descriptor prior to performing any reads to or writes from the stream.

The behavior and disposition of a stream may be influenced by the `Options` argument. `Options` is a list comprising one or more of the following terms:

`type(T)` – `T` may be either text or binary. This defines whether the stream is a text stream or a binary stream. At present, ALS Prolog makes no distinction between these two types.

`alias(Alias)` – `Alias` must be an atom. This option specifies an alias for the stream. If an alias is established, the alias may be passed in lieu of the stream descriptor to predicates requiring a stream handle.

`reposition(R)` – `R` is either `true` or `false`. `reposition(true)` indicates that the stream must be repositionable. If it is not, `open/3` or `open/4` will throw an error. `reposition(false)` indicates that the stream is not repositionable and any attempt to reposition the stream will result in an error. If neither option is specified, the stream will be opened as repositionable if possible. A program can find out if a stream is repositionable or not by calling `stream_property/2`.

`eof_action(Action)` – `Action` may be one of `error`, `eof_code`, or `reset`. `Action` instantiated to `error` indicates that an existence error should be triggered when a stream attempts to read past end-of-file. The default action is `eof_code` which will cause an input predicate reading past end-of-file to return a distinguished value as the output of the predicate (either `end_of_file`, or `-1`). Finally, `Action` instantiated to `reset` indicates that the stream should be reset upon end-of-file.

`snr_action(Action)` – `Action` may be one of `error`, `snr_code`, or `wait`. As with `eof_action`, `Action` instantiated to `error` will generate an existence error when an input operation attempts to read from a stream for which no input is ready. `snr_code` will force the input predicate to return a distinguished code when the stream is not ready. This code will be either `-2` or the atom `stream_not_ready`. The default action is `wait` which will force the input operation to wait until the stream is ready.

`buffering(B)` – `B` is either `byte`, `line`, or `block`. This option applies to streams open for output. If byte buffering is specified, the stream buffer will be flushed (actually written out) after each character. Line buffering is useful for streams which interact with a user; the buffer is flushed when a newline character is put into the buffer. Block buffering is the default; the buffer is not flushed until the block is full or until a call to `flush_output/1` is made.

`bufsize(Size)` – `Size` must be a positive integer. The `Size` parameter indicates the size of buffer to allocate the associated stream. The default size should be adequate for most streams.

`prompt_goal(Goal)` – `Goal` is a ground callable term. This option is used when opening an input stream. Goal will be run each time a new buffer is read. This option is most useful when used in conjunction with opening an output stream where a prompt should be written to whenever new input is required from the input stream.

`maxdepth(Depth)` – `Depth` is a positive integer. This option when specified for an output stream sets the default maximum depth used to write out a term with `write_term/3`, et. al. Explicit options to `write_term/3` may be used to override this option.

`depth_computation(DC)` – `DC` should be either `flat` or `nonflat`. This option indicates the default mechanism to be used for write_term to compute the depth of a term. `flat` indicates that all arguments in a list or structured term should be considered to be at the same depth. `nonflat` indicates that each successive element of a structured term or list is at depth one greater than its predecessor.

`line_length(Length)` – `Length` is a positive integer. This option is used to set the default line length associated with the stream. Predicates which deal with term output use this parameter to break the line at appropriate points when outputting a term which will span several lines. Explicit options to `write_term/3` may be used to override this option.

`write_eoln_type(Type) -` allows control over which end-of-line (`eoln`) characters are output by `nl/1.` The values for `Type` and the corresponding `eoln` characters are: `cr ("\r"), lf ("\n"),` and `crlf ("\r\n").` The default $^{\text{Type}}$ is determined by the operating system: MacOS (`cr`), Unix (`lf`), and Win32/DOS (`crlf`).

`read_eoln_type(Type) -` determines what `read/2` and `get_line/3` recognize as an end-of-line. The values for Type and the corresponding ends-of-line are: `cr` - carriage return (`"\r"`), `lf` - line feed ("`\n`"), `crlf` - carriage return followed by a line feed ("\r`\n`") , `universal` - indicates that any of the end-of-line types (`cr, lf, crlf`) should be interpreted as an end-of-line. The default is `universal` since this allows the correct end-of-line interpretation for text files on all operating systems.

EXAMPLES

Open file named example.dat for write access, write a term to it and close it.

```
?- open('example.dat',write,S), writeq(S,
example(term)),
?-_  put_char(S,'.'), nl(S), close(S).
```

```
S = stream_descriptor('',closed,file,'example.dat',
    [noinput|output], true,2,0,0,0,0,true,0,
   wt_opts(78,40000,flat),[],true,text,eof_code,0,0)
```

Open file named example.dat for read access with alias example.

```
?- open('example.dat',read,_,[alias(example)]).
```

Read a term from stream with alias example.

```
?- read(example,T).
```

```
T = example(term)
```

Read another term from with stream with alias example.

```
?- read(example,T).
```

```
T = end_of_file
```

```
Close the stream aliased example.
```

```
?- close(example).
```

The following procedure will open two source/sinks, one for read access, the other for write access. It will read one term from the source and write it to the sink. Finally, it will close both streams.

```
copy_one_term(In,Out) :-
       open(In,read,SI),
       open(Out,write,SO),
       read(SI,Term),
       writeq(SO,Term),
       put_char(SO,'.'),
       nl(SO),
       close(SI),
       close(SO).
```

Call copy_one_term/2 to copy the term in example.dat to a character list stream.

```
?- copy_one_term('example.dat', char_list(L)).

L = [e,x,a,m,p,l,e,'(',t,e,r,m,')',.,'\n']
```

Call `copy_one_term/2` to overwrite 'example.dat' with a new term specified by a character code list stream.

```
?-
copy_one_term(code_list("new(term).\n"),'example.dat
').
```

Call copy_one_term/2 to read a term from example.dat and put it into an atom stream.

```
?- copy_one_term('example.dat',atom(A)).

A = 'new(term).\n'
```

Attempt to open a stream with read access which does not exist.

```
?- open(foobar,read,S).

Error: The open operation is not permitted on the
source_sink object foobar.
  - Goal:          sio:open(foobar,read,_A,?)
  - Throw pattern: error(

permission_error(open,source_sink,foobar),

[sio:open(foobar,read,_A,?)])
```

ERRORS

`Name`, `Mode`, or `Options` is a variable
  ——> `instantiation_error.`

`Name` does not refer to either a variable or a source/sink
  ——> `domain_error(source_sink,Name).`

`Mode` is neither a variable nor an atom
  ——> `type_error(atom,Mode).`

`Mode` is an atom, but not a valid I/O mode for the given source/sink
  ——> `domain_error(io_mode,Mode).`

`Stream` is not a variable
  ——> `type_error(variable,Stream).`

`Options` is neither a variable nor a list
  ——> `type_error(list,Options).`

`Options` is a list with a variable element
  ——> `instantiation_error.`

`Options` is a list with element E which is not a valid stream option
  ——> `domain_error(stream_option,E).`

`Name` specifies a valid source/sink, but can not be opened. If `Name` refers to a file, the file may not exist or the protection on the file or containing directory might be set to be incompatible with the open mode

——> `permission_error(open,source_sink,Name).`

`Options` contains an element `alias(A)` and `A` is already associated with another stream

——> `permission_error(open,source_sink,alias(A)).`

`Options` contains an element `reposition(true)` and it is not possible to reposition a stream corresponding to the source/sink `Name`.

——>
`permission_error(open,source_sink,reposition(true))`
`.`

NOTES

The structured term comprising a stream descriptor is visible to the programmer. The programmer should not directly use the stream descriptor to learn of properties or attributes associated with the stream or otherwise rely on the representation of stream descriptors. Use `stream_property/2` to examine the properties associated with a stream.

The DEC-10 compatibility predicates `see/1` and `tell/1` are defined in terms of `open/4`. When a stream is opened with either of these predicates it is assigned an alias which is the name of the source/sink. Thus the single argument to `see/1` and `tell/1` may be considered to be both the name of the stream and an alias for the stream.

SEE ALSO

`close/1, current_input/1, current_output/1, flush_input/1, flush_output/1, stream_property/2, read_term/3, write_term/3, get_char/1, put_char/1, set_stream_position/2,` *User Guide (Prolog I/O).*

NAME

`peek_char/1` - obtain char from stream
`peek_char/2`

FORMS

peek_char(Char)

peek_char(Stream_or_alias, Char)

DESCRIPTION

`peek_char(Char)` unifies `Char` with the next character obtained from the default input stream. However, the character is not consumed.

`peek_char(Alias_or_Stream, Char)` unifies `Char` with the next character obtained from the stream associated with `Alias_or_Stream`. However, the character is not consumed.

NAME

`poll/2` – See if I/O is possible

FORMS

```
poll(Stream_or_Alias, TimeOut)
```

DESCRIPTION

`poll/2` will wait at most `TimeOut` microseconds and then succeed if a non-blocking I/O operation may be started on the stream associated with `Stream_or_Alias`. Failure will occur if the I/O operation would block.

ERRORS

`Stream_or_Alias` is a variable

 ——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

 ——> `domain_error(stream_or_alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream

 ——> `existence_error(stream,Stream_or_Alias).`

`TimeOut` is a variable

 ——> `instantiation_error.`

`TimeOut` is not an integer

 ——> `type_error(integer,TimeOut).`

NOTES

Note that an input operation such as `read/2` may block anyway if there is insufficient input to syntactically complete the term and the terminating full-stop. It is possible for `poll` to succeed when only the first character of the term is ready. The `open/[3,4]` option, `snr_action`, is better used for situations where reading a term from a stream which might not be ready is desirable.

SEE ALSO

open/[3,4],*User Guide (Prolog I/O).*

NAME

`printf/1`	– print out a string to the current output
`printf/2`	– print out a string with arguments
`printf/3`	– print out a string with a format and arguments
`printf/4`	– print out a string with a format, arguments, and options

FORMS

```
printf(Format)
printf(Format,ArgList)
printf(Steam_or_Alias,Format,ArgList)
printf(Stream_or_Alias,Format,ArgList,WriteOptions)
```

DESCRIPTION

`printf/2` takes a format string and a list of arguments to include in the format string. `printf/1` is the same as `printf/2` except no argument list is given. The following is a list of the special formatting possible within the format string:

`\n` – prints a newline (same as `nl/0`)

`\t` – prints a tab character

`\\` – prints a backslash

`\%` – prints a percent sign

`%c` – prints the corresponding Prolog character (atom) in the argument list

`%d` – prints the corresponding decimal number in the argument list

`%s` – prints the corresponding Prolog string in the argument list

`%t` – prints the corresponding Prolog term in the argument list (same as `write/1`)

All other characters are printed as they appear in the format string.

Using `printf` is generally much easier than using the equivalent `write/1`, `put/1`, and `nl/0` predicates because the whole message you want to print out can be done by one call to `printf`.

EXAMPLES

```
?- printf("hello world\n").
hello world

yes.
?- printf("Letters: %c%c%c\n",[a,b,c]).
Letters: abc

yes.

?- printf("Contents: %t, Amount: %d\n",
        [pocket(keys,wallet,watch), 3]).
Contents: pocket(keys,wallet,watch), Amount: 3
yes.
```

SEE ALSO

`nl/0`, `put/1`, `write/0`, User Guide (Prolog I/O), [Unix/C Reference Manuals: `printf(3S)`].

NAME

`procedures/4`	– retrieves all Prolog-defined procedures
`all_procedures/4`	– retrieves all Prolog- or C-defined procedures
`all_ntbl_entries/4`	– retrieves all name table entries

FORMS

```
procedures(Module,Pred,Arity,DBRef)
all_procedures(Module,Pred,Arity,DBRef)
all_ntbl_entries(Module,Pred,Arity,DBRef)
```

DESCRIPTION

For all three of these 4-argument predicates, the system name table is searched for an entry corresponding to the triple `(Module,Pred,Arity)`. If such an entry is found, the name table entry is accessed and `DBRef` is unified with the database reference of the procedure's first clause.

`procedures/4` only considers procedures defined in Prolog; `all_procedures/4` considers just procedures defined in either Prolog or C; `all_ntbl_entries/4` considers all name table entries. If the triple `(Module,Pred,Arity)` is not completely specified, all matching name table entries of the appropriate sort are successively returned.

EXAMPLES

```
?- all_procedures(builtins,P,A,DB).
P = nonvar
A = 1
DB = 0;

P = edit2
A = 1
DB = '$dbref'(404,21,24,0);

P = see
A = 1
```

```
DB = 0;

P = gc
A = 0
DB = 0

yes.
```

NAME

`'$procinfo'/5`	– retrieves information about the given procedure
`'$nextproc'/3`	– retrieves the next procedure in the name table
`'$exported_proc'/3`	– checks whether the given procedure is exported
`'$resolve_module'/4`	– finds the module which exports the given procedure

FORMS

```
'$procinfo'(NTblIndex,Module,Pred,Arity,DBRef)
'$nextproc'(PreNTblIndex,Flag,NTblIndex)
'$exported_proc'(Module,Pred,Arity)
'$resolve_module'(Module,Pred,Arity,ImportedFrom)
```

DESCRIPTION

Given a name table index (`NTBlIndex`), `'$procinfo'/5` returns the module name, the predicate name, the arity, and the database reference associated with that name table entry. If the given name table entry is a Prolog-defined procedure (as opposed to a C- or assembler-defined procedure), the returned database reference is the database reference of the procedure's first clause. If the procedure associated with `NTblIndex` is not a Prolog-defined procedure, 0 will be returned as the database reference. If `NTblIndex` is an uninstantiated variable, and `Module`, `Pred` and `Arity` are bound values, the name table entry of `Pred/Arity` in module `Module` is accessed.

`'$nextproc'/3` returns the name table index `NTblIndex` of the next name table entry following the input name table entry `PreNTblIndex`. If `PreNTblIndex` is -1, the first name table entry index is returned. The argument `Flag` determines the type of the next name table entry to be chosen, as follows.

Table 18:

`Flag=0`	The index of the name table entry of the next Prolog-defined procedure is returned.
`Flag=1`	The index of the name table entry of the next Prolog- or C-defined procedure is returned.
`Flag=2`	The index of the next name table entry, regardless of its type, is returned.

Table 19. Argument Flags.

For `’$exported_proc’/3`, if the procedure whose module name `Module`, predicate name `Pred`, and `Arity` are given is exported, `’$exported_proc’/3` succeeds; otherwise it fails.

For `’$resolve_module’/4`, if the given procedure is not defined in the given module `Module`, `ImportedFrom` is unified with the name of the module from which the given module `Module` imports the procedure `Pred/Arity`.

EXAMPLES

```
?-[nrev].
Consulting nrev ...
nrev consulted
yes.

?- ’$exported_proc’(user,nrev,2).
no.

?- [user].
Consulting user ...
  export nrev/2.
  user consulted
yes.

?- '$exported_proc'(user,nrev,2).
yes.

?- '$resolve_module'(user,append,3,ImportedFrom).
ImportedFrom = user
yes.
```

NAME

`put/1`	- write out a character
`tab/1`	- prints out a specified number of spaces

FORMS

```
put(Char)
tab(N)
```

DESCRIPTION

If `Char` is bound to an integer within the range 0–255, `put/1` will write out the character whose ASCII code is `Char` to the current output stream.

`tab/1` will write out `N` space characters (ASCII 32) to the standard output stream.

EXAMPLES

```
?- put(~(), tab(15), put(~)).
(                )
yes.
```

SEE ALSO

`nl/0`, User Guide (Prolog I/O),[Bowen 91, 7.8], [Clocksin 81, 5.2], [Bratko 86, 6.3].

NAME

`put_atom/1`	– output an atom to the current output stream
`put_atom/2`	– output an atom to a specific output stream

FORMS

```
put_atom(Atom)
put_atom(Stream_or_Alias, Atom)
```

DESCRIPTION

`put_atom/1` will write out the atom bound to `Atom` to the current output stream.

`put_atom/2` will write out the atom bound to `Atom` to the output stream associated with `Stream_or_Alias`.

EXAMPLES

```
?- put_atom(ice),put_atom(cream).
icecream
```

ERRORS

Stream_or_Alias is a variable

——> `instantiation_error.`

Stream_or_Alias is neither a variable nor a stream descriptor nor an alias

——>
`domain_error(stream_or_alias,Stream_or_Alias).`

Stream_or_Alias is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

Stream_or_Alias is not an output stream

——>
`permission_error(output,stream,Stream_or_Alias).`

Atom is a variable

——> `instantiation_error.`

Atom is neither a variable nor an atom

——> `type_error(atom,Atom).`

NAME

`put_char/1`	– output a character to the current output stream
`put_char/2`	– output a character to a specific output stream

FORMS

```
put_char(Char)
put_char(Stream_or_Alias,Char)
```

DESCRIPTION

`put_char/1` will write out the character bound to `Char` to the current output stream.

`put_char/2` will write out the character bound to `Char` to the output stream associated with `Stream_or_Alias`.

EXAMPLES

```
?- put_char('\t'),put_char(h),put_char(o),
?-_put_char(w),put_char(d),put_char(y),nl.
        howdy
```

ERRORS

Stream_or_Alias is a variable

——> `instantiation_error.`

Stream_or_Alias is neither a variable nor a stream descriptor nor an alias

——>
`domain_error(stream_or_alias,Stream_or_Alias).`

Stream_or_Alias is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

Stream_or_Alias is not an output stream

——>

`permission_error(output,stream,Stream_or_Alias).`

Char is a variable

——> `instantiation_error.`

Char is neither a variable nor a character

——> `type_error(character,Char).`

SEE ALSO

`get_char/1`, `put_code/1`, `open/4`, `close/1`, `char_code/2`, `nl/1`, ***User Guide (Prolog I/O).***

.

NAME

`put_code/1`	– output a character code to the current output stream
`put_code/2`	– output a character code to a specific output stream

FORMS

```
put_code(Char)
put_code(Stream_or_Alias,Code)
```

DESCRIPTION

`put_code/1` will write out the character code bound to `Char` to the current output stream.

`put_code/2` will write out the character code bound to `Char` to the output stream associated with `Stream_or_Alias`.

EXAMPLES

```
?- put_code(0'\t), put_code(0'h), put_code(0'o),
put_code(0'w),
?-_  put_code(0'd), put_code(0'y), put_code(0'\n).
        howdy
```

ERRORS

Stream_or_Alias is a variable

——> `instantiation_error.`

Stream_or_Alias is neither a variable nor a stream descriptor nor an alias

——>
`domain_error(stream_or_alias,Stream_or_Alias).`

Stream_or_Alias is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

Stream_or_Alias is not an output stream

——>
`permission_error(output,stream,Stream_or_Alias).`

Code is a variable

——> `instantiation_error.`

Code is neither a variable nor a character

——> `type_error(character,Code).`

SEE ALSO

`get_code/1`, `put_char/1`, `open/4`, `close/1`, `char_code/2`, `nl/1`, ***User Guide (Prolog I/O)***.

NAME

`put_string/1`	– output a string to the current output stream
`put_string/2`	– output a string to a specific output stream

FORMS

```
put_string(String)
put_string(Stream_or_Alias, String)
```

DESCRIPTION

`put_string/1` will write out the string bound to `String` to the current output stream.

`put_string/2` will write out the string bound to `String` to the output stream associated with `Stream_or_Alias`.

EXAMPLES

```
?- put_string(“ice”),put_string(“cream”).
icecream
```

ERRORS

Stream_or_Alias is a variable

——> `instantiation_error.`

Stream_or_Alias is neither a variable nor a stream descriptor nor an alias

——>
`domain_error(stream_or_alias,Stream_or_Alias).`

Stream_or_Alias is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

Stream_or_Alias is not an output stream

——>
`permission_error(output,stream,Stream_or_Alias).`

String is a variable

——> `instantiation_error.`

String is neither a variable nor a string

——> `type_error(string,String).`

NAME

`read/1`	– read a term from the current input stream
`read/2`	– read a term from specified stream
`read_term/2`	– read term from current input with options
`read_term/3`	– read term from specified stream with options

FORMS

```
read(Term)
read(Stream_or_Alias,Term)

read_term(Term,Options)
read_term(Stream_or_Alias,Term,Options)
```

DESCRIPTION

These predicates are used to read a term from a stream using the operator declarations in effect at the time of the read. The end of the term read from the stream is indicated by a fullstop token appearing in the stream. The fullstop token is a period (`'.'`) followed by a newline, white space character, or line comment character. If the stream is positioned so that there are no more terms to be read and the stream has the property `eof_action(eof_code)`, then `Term` will be unified with the atom `end_of_file`.

`read/1` reads a term from the current input stream and unifies it with `Term`.

`read/2` reads a term from the input stream specified by `Stream_or_Alias` and unifies it with `Term`.

`read_term/2` reads a term from the current input stream with options `Options` (see below) and unifies the term read with `Term`.

`read_term/3` reads a term from the input stream specified by `Stream_or_Alias` and unifies it with `Term`. The options specified by `Options` are used in the process of reading the term.

`read_term/2` and `read_term/3` take the parameter `Options` which is a list of options to `read_term`. These options either affect the behavior of `read_term` or are used to retrieve additional information about the term which was read.

The following options supported by ALS Prolog are options specified by the March '93 ISO Prolog Standard..

`variables(Vars)` – After reading a term, `Vars` shall be a list of the variables in the term read, in left-to-right traversal order.

`variable_names(VN_list)` – After reading a term, `VN_list` will be unified with a list of elements of the form `V=A` where `V` is a variable in the term and `A` is an atom representing the name of the variable. Anonymous variables (variables whose name is '_') will not appear in this list.

`singletons(VN_list)` – After reading a term, `VN_list` will be unified with a list of elements of the form `V=A`, where `V` is a variable occurring in the term only once and `A` is an atom which represents the name of the variable. Anonymous variables will not appear in this list.

The following option does not conform to the standard, but is supported by ALS Prolog.

`attach_fullstop(Bool)` – `Bool` is either `true` or `false`. The default is `false`. When `Bool` is `true`, a fullstop is inserted before the end of the stream. This option is most useful for reading single terms from atom, character, and character code streams.

EXAMPLES

```
?- read(Term).
[ +(3,4), 9+8 ].

Term = [3+4,9+8]


?- read_term(Term, [variables(V),
variable_names(VN), singletons(SVN)]).
f(X,[Y,Z,W],g(X,Z),[_,U1,_,U2]).

Term = f(_A,[_B,_C,_D],g(_A,_C),[_E,_F,_G,_H])
```

```
V = [_A,_B,_C,_D,_E,_F,_G,_H]
VN = [_A = 'X',_B = 'Y',_C = 'Z',_D = 'W',_F = 'U1',_H
= 'U2']
SVN = [_B = 'Y',_D = 'W',_F = 'U1',_H = 'U2']


?- open(atom('[X,2,3]'),read,S),
?-_  read_term(S, Term, [attach_fullstop(true)]),
?-_  close(S).

S =
stream_descriptor('',closed,atom,atom('[X,2,3]'),

[input|nooutput],false,3,'[X,2,3]',7,7,0,true,0,

wt_opts(78,40000,flat),[],true,text,eof_code,0,0)
Term = [_A,2,3]


?- read_term(user_input, Term, not_an_option_list).

Error: Argument of type list expected instead of
not_an_option_list.
- Goal:          sio:
read_term(user_input,_A,not_an_option_list)
- Throw pattern:
error(type_error(list,not_an_option_list),
                     [sio:read_term(user_input,_A,
                         not_an_option_list)])



?- read(X).
foo bar.

foo bar.
```

```
    ^
Syntax error: ’$stdin’, line 17: Fullstop (period)
expected
foo.
X = foo
```

ERRORS

`Stream_or_Alias` is a variable
    ——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias
    ——>
`domain_error(stream_or_Alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream
    ——>
`existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` is not an input stream
    ——>
`permission_error(input,stream,Stream_or_Alias).`

`Options` is a variable
    ——> `instantiation_error.`

`Options` is neither a variable nor a list
    ——> `type_error(list,Option).`

`Options` is a list an element of which is a variable
    ——> `instantiation_error.`

`Options` is a list containing an element `E` which is neithera variable nor a valid read option
    ——> `domain_error(read_option,E).`

One or more characters were read, but they could not be parsed as a term using the current set of operator definitions
    ——> `syntax_error.` [This does not happen now; see notebelow]

The stream associated with `Stream_or_Alias` is at the end of the stream and the stream has the property `eof_action(error)`

——>

```
existence_error(past_end_of_stream,Stream_or_Alias)
.
```

The stream associated with `Stream_or_Alias` has no input ready to be read and the stream has the property `snr_action(error)`

——>

```
existence_error(stream_not_ready,Stream_or_Alias).
```

NOTES

The ISO Prolog Standard requires that an error be thrown when there is a syntax error in a stream being read. The default action at the present time for ALS Prolog is to print out an error message describing the syntax error and then attempt to read another term. This action is consistent with the behavior of older DEC-10 compatible Prologs. It is expected that ALS Prolog will eventually comply with the standard in this respect.

SEE ALSO

`write/[1,2]`, `write_term/[2,3]`, `open/4`, `close/1`, `get_char/[1,2]`, `get_code/[1,2]`, User Guide (Prolog I/O), [Bowen 91, 7.8], [Sterling 86, 12.2], [Bratko 86, 6.2.1], [Clocksin 81, 5.1].

NAME

`recorda/3`	– records item in internal term database
`recordz/3`	– records item in internal term database
`recorded/3`	– retrieves item from internal term database

FORMS

```
recorda(Key,Term,Ref)
recordz(Key,Term,Ref)
recorded(Key,Term,Ref)
```

DESCRIPTION

`Key` may be an atom or a compound term, but in the latter case, only its functor is significant. Both predicates will fail in all other cases of `Key`. `recorda(Key,Term,Ref)` enters `Term` into the internal term database at the first item associated with `Key`, and returns a database reference `Ref`. `recordz(Key,Term,Ref)` enters `Term` as the last item associated with `Key`.

`recorded(Key,Term,Ref)` searches the internal term database for an item `Term` associated with `Key` such that the associated database reference unifies with `Ref`.

EXAMPLES

```
?-recordz(sing, slowly, _),
  recorda(sing, sweetly, _),
  recorda(sing(along), loudly, _).
yes.

?-recorded(sing, Term, _).
Term = loudly;
Term = sweetly;
Term = slowly;
no.
```

NOTES

Provided for compatibility; these predicates are defined by:

```
recorda(Key,Term,Ref) :-
   rec_getkey(Key,KeyFuncotr),
   asserta(recorded(KeyFuncotr,Term),Ref).

recordz(Key,Term,Ref) :-
   rec_getkey(Key,KeyFunctor),
   assertz(recorded(KeyFunctor,Term),Ref).


recorded(Key,Term,Ref) :-
   rec_getkey(Key,KeyFunctor),
   clause(recorded(KeyFunctor,Term), true, Ref).

rec_getkey(Key, Key) :- atomic(Key), !.
rec_getkey(S,Key) :- functor(S,Key,_).
```

NAME

`repeat/0` – always succeed upon backtracking

FORMS

```
repeat
```

DESCRIPTION

`repeat/0` always succeeds, even during backtracking. This behavior is useful for implementing loops which repeatedly perform some side-effect. `repeat/0` is defined by the following clauses:

```
repeat.
repeat :- repeat.
```

EXAMPLES

The following procedure will repeat forever, reading in an expression and printing out its value.

```
loop :-
    repeat,
    read(Expression),
    Value is Expression,
    write('Value = '), write( Value ), nl,
    fail.
```

SEE ALSO

`fail/0`,

[Sterling 86, 12.5], [Bratko 86, 7.5], [Clocksin 81, 6.6].

NAME

`retract/1`	– removes a clause from the database
`retract/2`	– removes a clause specified by a database reference
`erase/1`	– removes a clause from the database

FORMS

```
retract(Clause)
retract(Clause,DBRef)
erase(DBRef)
```

DESCRIPTION

When `Clause` is bound to an atom or a structured term, the current module is searched for a clause that will unify with `Clause`. When a matching clause is found in the database, `Clause` is unified with the structure corresponding to the clause. The clause is then removed from the database.

`retract/2` additionally unifies `DBRef` with the database reference of the clause.

`erase/1` removes the clause associated with `DBRef` from the database. Note that `erase(DBRef)` should never be called following `retract(Clause,DBRef)` since at that point `DBRef` is no longer a valid database reference.

`retract/1` and `retract/2` will repeatedly generate and remove clauses upon backtracking. `:/2` can be used to specify which module should be searched.

EXAMPLES

The following example shows how `retract/1` and `retract/2` can be used to get rid of all the comic book heroes that live in our modules. First we create all the heroes by consulting `user`. Then we get rid of `hero(spiderman)` by using a simple call to `retract/1`:

```
?- [user].
Consulting user.
```

```
  hero(spiderman).
  hero(superman).
  hero(batman).

  module girls.
  hero(superwoman).
  endmod.
^D user consulted.

yes.
?- retract(hero(spiderman)).

yes.
```

In the next example, we show what heroes are left by using the `listing/1` procedure. After that, we remove `hero(superman)`with a `retract/2` call. The old database reference to the man of steel is instantiated to `Ref`.

?- listing(hero/1).

```
% user:hero/1
hero(superman).
hero(batman).
% girls:hero/1
hero(superwoman).

yes.
?- retract(hero(superman),Ref).
Ref = '$dbref'(5208,15,2384,1)

yes.
```

In this next example, we use `clause/3` to find the database reference of `hero(batman)`. After this, we use the database reference in a `retract/2` call to remove `hero(batman)` from the database. Note that the `Clause` argument for `retract/2` is uninstantiated in this call. The clause that was removed is instantiated to `Clause`, after the call to `retract/2` has succeeded.

```
?- clause(hero(batman),Body,Ref).
```

```
Body = true
Ref = '$dbref'(5052,15,2384,2)

yes.
?- retract(Clause,'$dbref'(5052,15,2384,2)).
Clause = hero(batman)
yes.
```

In the following example, we list the heroes left in the database. Only `hero(superwoman)` is left, but she's in a different module. However, using the `Mod:Goal` construct, we can remove her too:

```
?- listing(hero/1).
% girls:hero/1
hero(superwoman).

yes.
?- girls:retract(hero(X)).
X = superwoman

yes.
?- listing(hero/1).

yes.
```

As the last call to `listing/1` shows, there are no more heroes left in the database. (Who knows what evil may be lurking in the garbage collector though!)

SEE ALSO

`clause/1`,

[Bowen 91, 7.3], [Clocksin 81, 6.4], [Bratko 86, 7.4], [Sterling 86, 12.2].

NAME

`reverse/2`	– list reversal
`dreverse/2`	– determinate list reversal

FORMS

```
reverse(List1,List2)
dreverse(List1,List2)
```

DESCRIPTION

`reverse/2` succeeds when `List2` can be unified with the result of reversing `List1`. `dreverse/2` is the determinate version of `reverse/2`.

EXAMPLES

```
?- reverse([a,b,c], List2).
List2 = [c,b,a]

yes.
```

NOTES

Defined by the following clauses:

```
reverse(List,Rev) :-
       reverse(List,[],Rev).

reverse([],Rev,Rev).
reverse([A|Rest],SoFar,Rev) :-
       reverse(Rest,[A|SoFar],Rev).

dreverse(List,Rev) :-
       dreverse(List,[],Rev).

dreverse([],Rev,Rev) :- !.
dreverse([A|Rest],SoFar,Rev)
```

```
        :- dreverse(Rest,[A|SoFar],Rev).
```

NAME

`rexec/2` – Execute an operating system command, possibly remotely.

FORMS

```
rexec(Command,Options)
```

DESCRIPTION

`rexec/2` is an interface to the rexec system call which may be used to run commands on remote machines. When remote execution is not desired, `fork` and `exec` (Unix system calls) are used to run the command on the local machine. Command should be an atom representing the command to run. Options is a list containing zero or more of the following forms:

`host(HostName)` – execute the command on the machine named by `HostName`.

`username(User)` – run the command as user `User`.

`password(Password)` – provides the password for authentication purposes. If no password is supplied, you will be prompted for one (by the `rexec` daemon).

`rstream(Stream,OpenOpts)` – designates the input stream to read the output of the command from. This stream will be connected to the standard output of the command. `Stream` will be bound to a stream descriptor. `OpenOpts` is a list containing options suitable for a call to `open/4`.

`wstream(Stream,OpenOpts)` – designates the output stream to write to. This output stream will be connected to standard input of the command.

estream(Stream,OpenOpts) – designates the input stream for use in obtaining error messages from the command. This stream will be connected to standard error for the command.

If any of host, username, or password are specified, rexec/2 will attempt to contact the rexec daemon to remotely run the command on the specified machine. The remote execution daemon, rexecd, requires authentication. This means that either a username and password must be supplied in the program (with the username/1 and password/1 forms), interactively, or via the .netrc file. Not all remote execution daemons support authentication via the .netrc file. See your local system documentation for information about the .netrc file.

If none of host, username, or password are specified, then rexec/2 will use the traditional fork and exec mechanism to run the command on the local machine. If remote execution is still desired, but the rexec daemon's authentication mechanisms deemed too odious, then rsh (running on the local machine) may be used to run a command on a remote machine.

EXAMPLES

The following procedure will call the unix word count program, wc, to determine the length of an atom.

```
slow_atom_length(A,Len) :-
       rexec('wc -c',
              [rstream(RS,[]),wstream(WS,[])]),
       write(WS,A),
       close(WS),
       read_term(RS,Len,[attach_fullstop(true)]),
       close(RS).

  ?- slow_atom_length('The rain in Spain', Len).

  Len = 17

  yes.
```

The version of `slow_atom_length/2` above assumes one is running on a Unix machine and calls wc running on the same machine. The version below, `slow_atom_length/3` , will work on any system which supports sockets (Unix workstations, Windows 95 with WinSock, Macintosh):

```
rstrlen(Host,A,Len) :-
    rexec('wc -c',
            [host(Host),rstream(RS,[]),
                 wstream(WS,[])]),
    write(WS,A),
    close(WS),
    read_term(RS,Len,[attach_fullstop(true)]),
    close(RS).
```

Here is an interaction running over the internet:

```
?- rstrlen('bongo.cs.anywhere.edu','abcdef',X).
Name (bongo.cs.anywhere.edu:ken): mylogin
Password (bongo.cs.anywhere.edu:ken): <mypasswd>

X = 6
```

NOTES

This function is not yet very consistent with regards to error handling. Some errors will be thrown, while others will print a diagnostic. Other errors will cause failure. This functionality will be cleaned up in a later release.

SEE ALSO

open/[3,4].

NAME

`save_image/2` – package an application

FORMS

`save_image(NewImage,Options)`

DESCRIPTION

`save_image/2` is called to package up the Prolog's code areas, symbol table, and module information into a single image. Any Prolog library code which the image depends on will not by default be packaged into the image.

`NewImage` should be bound to an atom which represents the pathname to the new image to be created. `Options` is a list of options which control the disposition and startup characteristics for the new image. The forms which may be on an options list to `save_image/2` are:

`start_goal(SGoal)` – `SGoal` is a goal which is accessible from the user module. This goal is run in place of the current starting goal (usually the Prolog shell) when the application starts up. If the `start_goal(SGoal)` form is not specified in the options list, then the current starting goal is retained as the starting goal for the new image.

`init_goals(IGoal)` – `IGoal` is either a single goal or a conjunction of goals to be run prior to the starting goal (see above). `IGoal` will be executed after the initialization goals added by previous packages including the ALS Prolog system itself. This form provides a mechanism for performing initializations which the present environment requires and would be required should any packagesbe built upon the newly saved image.

`libload(Bool)` – `Bool` is either `true` or `false`. If `true`, the Prolog library is loaded as part of the package. This is necessary since the Prolog library is demand loaded and may not be part of the development environment when the image is saved. If `false`, the Prolog library is not loaded as part of the package. Once created, however, the library may still be (demand) loaded by the new image, provided the library files are still accessible to the new image. In general, applications which require the Prolog library and will leave the machine on which the development environment exists should specify Bool as `true`. Applications which may need the library but will be run on the same machine as the development environment can specify Bool as `false` if it is necessary to keep the space requirements for the new image as small as possible.

`select_lib(FilesList)` – `FilesList` is a list library file names from the Prolog library. Each of the listed library files is loaded as part of the package.

The process of creating a new image consists of the following steps:

1. Process the options, changing the starting goal, extending the initialization goals, or forcefully loading the Prolog library as specified by the options.
2. Create a saved code state which is put into a temporary directory. The directory in which this saved code state is put may be influenced by changing the TMPDIR variable.
3. Merge the saved code state and the current Prolog image together into a new image file.

EXAMPLES

```
?- save_image(hello, [start_goal(printf('Hello
world\n',[]))]).
Executing /max4/kev/merge3/M88k/Mot-SVR4/obj/./
alsdir/als-mics /max4/kev/merge3/M88k/Mot-SVR4/obj/
alspro /var/tmp/aptAAAa000un hello

yes.
?- halt.
max:scratch$ hello
Hello world
max:scratch$
```

NOTES

The als-mics program is required to merge the code state with a working ALS Prolog image. If this program does not exist or is inaccessible, an image will not be saved. The place where als-mics is searched for can be obtained by entering the following query:

```
?- builtins:sys_searchdir(Where).
```

Where will be bound to the directory where `save_image` expects to find the als-mics program.

Global variable values and database assertions dealing with environmental issues should be initialized (or reinitialized) via a goal passed to `init_goals`.

`save_image/2` prints diagnostic messages when something goes wrong. This procedure will eventually be updated to throw errors in a manner compatible with the standard.

NAME

`see/1`	– sets the current input stream
`seeing/1`	– returns the name of the current input stream
`seen/0`	– closes the current input stream

FORMS

```
see(File)
seeing(File)
seen
```

DESCRIPTION

`see/1` sets the current input stream to the file named `File`. If `File` is already open for input, the existing file descriptor will be used. Otherwise, a new file descriptor will be allocated, and input operations will start at the beginning of the file.

`seeing/1` unifies `File` with the name of the current input stream. If no stream has been explicitly opened by `see/1`, then file will be unified with the atom `user`.

`seen/0` closes the current input stream and deallocates its file descriptor. The current input stream is then set to the special file *user*. *user* is the name of the default input stream which is normally connected to the keyboard. The special file user is always present, and `seen/0` can never close it. Although `seen/0` can never close the file user, `seen/0` will reset the user I/O descriptor as follows. If a `read/n` has been executed from user, and if an end of file character sequence is encountered (`Control-D` on Unix or the Mac, or `Control-Z` on Win32/DOS and on the Mac), then the `read/n` returns `end_of_file` and a subsequent `seen/0` on user will reset the I/O descriptor for user so that the EOF condition is no longer present.

EXAMPLES

The following program will

- preserve the current input stream

- open a file
- read one term from it
- restore the previous input stream

```
firstTerm(File,Term) :-
       seeing(CurrentInput),
       see(File),
       read(Term),
       seen,
       see(CurrentInput).
```

SEE ALSO

`see/1`, `seeing/1`, `seen/1`, `open/3`, `open/4`, `close/1`, `close/2`, User Guide (Prolog I/O), [Bowen 91, 7.8], [Clocksin 81, 5.4], [Bratko 86, 6.1].

NAME

`set_input/1`	– set current input stream
`set_output/1`	– set current output stream

FORMS

```
set_input(Stream_or_Alias)
set_output(Stream_or_Alias)
```

DESCRIPTION

For both forms `Stream_or_Alias` must be either a stream descriptor returned by a call to `open/3` or `open/4` or an alias created during a successful call to open. For the sake of the following discussion, let us suppose that `Stream` is `Stream_or_Alias` if `Stream_or_Alias` is a stream descriptor. If `Stream_or_Alias` is an alias, then let `Stream` be the stream associated with that alias.

`set_input/1` will set the stream associated with the current input stream to `Stream`. The current input stream is the stream that is read when predicates such as `get_char/1` or `read/1` are called. The current input stream may be retrieved by calling `current_input/1`.

`set_output/1` will set the stream associated with the current output stream to `Stream`. The current output stream is the stream which is written to when predicates such as `put_char/1` or `write/1` are called. The current output stream may be retrieved by calling `current_output/1`.

EXAMPLES

Suppose that the file "test" is comprised of the characters "abcdefgh\n".

Open the file "test" and set the current input stream to the stream descriptor returned from open.

```
?- open(test,read,S), set_input(S).

S =
stream_descriptor('',open,file,test,[input|nooutput]
,true,

4,0,0,0,0,true,0,wt_opts(78,40000,flat),[],true,
         text,eof_code,0,0)
```

Get two characters from the current input stream.

```
?- get_char(C1), get_char(C2).

C1 = a
C2 = b
```

Close the stream associated with "test".

```
?- current_input(S), close(S).

S =
stream_descriptor('',closed,file,test,[input|nooutpu
t],true,

4,0,0,0,0,true,0,wt_opts(78,40000,flat),[],true,
         text,eof_code,0,0)
```

ERRORS

`Stream_or_Alias` is a variable

——> `instantiation_error.`

`Stream_or_Alias` is not a variable nor a stream descriptor nor an alias

——>

`domain_error(stream_or_alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` is not an input stream (for set_input/1)

——>

`permission_error(input,stream,Stream_or_Alias).`

`Stream_or_Alias` is not an output stream (for `set_output/1`)

——>

`permission_error(output,stream,Stream_or_Alias).`

NOTES

`close/1` may also change the setting of the current input or output streams.

SEE ALSO

`current_input/1`, `open/3`, `close/1`, *User Guide (Prolog I/O).*

NAME

`set_line_length/2` – set length of line for output stream
`set_max_depth/2` – set maximum depth that terms will be written to
`set_depth_computation/2`– set method of computing term depth

FORMS

```
set_line_length(Stream_or_Alias,Length)
set_maxdepth(Stream_or_Alias, Depth)
set_depth_computation(Stream_or_Alias,Flat_or_Nonfla
t)
```

DESCRIPTION

`set_line_length/2` sets the default line length for the output stream associated with `Stream_or_Alias` to the integer value bound to `Length`. The default line length is an integer parameter used by `writeq/[1,2]`, `write_canonical/[1,2]`, and `write_term/[2,3]` to determine where line breaks should occur when outputting a term. A call to `write_term` may temporarily overide this parameter by specifying the `line_length` option in the write options list. The default line length may also be set at the time the stream is opened by specifying the `line_length` option in the options list to `open/[3,4]`.

`set_maxdepth/2` sets the default depth limit to which terms are output for the output stream associated with `Stream_or_Alias` to the integer value bound to Depth. The default depth limit is used by the term output predicates to determine the maximum depth to write to. This parameter may also be set at the time of an open with the appropriate open option and may be overridden in calls to `write_term/[3,4]` with the appropriate write option.

`set_depth_computation/2` sets the manner in which the depth of a term is computed for the output stream associated with `Stream_or_Alias` to the atomic value bound to `Flat_or_Nonflat`. As the name of the variable implies, `Flat_or_Nonflat` must be bound to one of the two atoms, `flat` or `nonflat`. If the depth computation method is `flat`, all arguments in a structured term and all list elements are considered to be at the same level. If the method is `nonflat`, then each subsequent structure argument or list element is considered to be at a depth one greater than the previous element.

EXAMPLES

```
?- set_line_length(user_output,20),
?-
_L=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,
y,z],
?-_write(L),nl.
[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
]

L = [a,b,c,d,e,f,g,
        h,i,j,k,l,
        m,n,o,p,q,
        r,s,t,u,v,
        w,x,y,z]

?- set_maxdepth(user_output,8),
?-_set_depth_computation(user_output,nonflat),
?-
_L=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,
y,z],
?-_write(L),nl.
[a,b,c,d,e,f,g,h,...]

L = [a,b,c,d,e,f,
          g,...]
```

ERRORS

`Stream_or_Alias` is a variable

——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

——>

`domain_error(stream_or_alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream
——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` is not an output stream
——>
`permission_error(output,stream,Stream_or_Alias).`

`Length`, `Depth`, or `Flat_or_Nonflat` is a variable
——> `instantiation_error.`

`Length` is not a variable or an integer
——> `type_error(integer,Length)`

`Length` is not an integer greater than four
——> `domain_error(line_length,Length)`

`Depth` is not a variable or an integer
——> `type_error(integer,Depth)`

`Depth` is an integer, but not a positive integer
——> `domain_error(positive_integer,Depth)`

`Flat_or_Nonflat` is neither a variable nor an atom
——> `type_error(atom,Flat_or_nonflat)`

`Flat_or_Nonflat` is an atom, but is neither flat nor nonflat
——>
`domain_error(depth_computation,Flat_or_nonflat)`

NOTES

Note in the above examples that `write/[1,2]` does not pay attention to the line length. It does however, observe the default maximum depth and the method for computing the depth.

SEE ALSO

`stream_property/2`, `open/4`, `write_term/3`, *User Guide (Prolog I/O)*.

NAME

`set_prolog_flag/2` – set value of a Prolog flag

FORMS

set_prolog_flag(Flag,Value)

DESCRIPTION

`set_prolog_flag/2` will modify the prolog flag, `Flag`, to the value, `Value`.

EXAMPLES

```
?- set_prolog_flag(undefined_predicate,fail).
```

SEE ALSO

current_prolog_flag/2

NOTES

The only flag implemented in ALS Prolog at this time is `undefined_predicate`.

ERRORS

`Flag` is a variable

    ——> `instantiation_error.`

`Value` is a variable

    ——> `instantiation_error.`

`Flag` is neither a variable nor an atom

    ——> `type_error(atom,Flag).`

`Value` is inappropriate for Flag

    ——> `domain_error(flag_value, Flag + Value)`

NAME

`setof/3`	– all unique solutions for a goal, sorted
`bagof/3`	– all solutions for a goal, not sorted
`findall/3`	– all solutions for a goal, not sorted
`b_findall/4`	– bound list of solutions for a goal, not sorted

FORMS

```
setof(Template,Goal,Collection)
bagof(Template,Goal,Collection)
findall(Template,Goal,Collection)
b_findall(Template,Goal,Collection, Bound)
```

DESCRIPTION

These predicates collect in the list `Collection`, the set of all instances of `Template` such that the goal, `Goal`, is provable. `Template` is a term that usually shares variables with `Goal`.

`setof/3` produces a `Collection` which is sorted according to the standard order with all duplicate elements removed. Both `bagof/3` and `findall/3` produce `Collection`s that are not sorted.

If there are no solutions to `Goal`, then `setof/3` and `bagof/3` will fail, whereas, `findall/3` unifies `Collection` with `[]`.

Variables that occur in `Goal` and not within `Template` are known as ***free variables***. `setof/3` and `bagof/3` will generate alternative bindings for free variables upon backtracking.

Within a call to `setof/3` or `bagof/3`, free variables can be existentially quantified in `Goal` by using the notation `Variable^Query`. This means that there exists a `Variable` such that `Query` is `true`.

The collection to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances of `Template` to contain variables, but in this case `Collection` will only provide an imperfect representation of what is actually an infinite collection.

setof/3 calls upon sort/2 to eliminate duplicate solutions from Collection, which seriously impacts its efficiency. In addition, even though bagof/3 leaves duplicate solutions, it still calls keysort/2.

findall/3 neither removes duplicates nor generates alternative bindings for free variables—it assumes that all variables occurring within Goal are existentially quantified. As a result, findall/3 is much more efficient than either setof/3 or bagof/3.

When Bound is a positive integer, b_findall/4 operates similarly to findall/3, except that it returns at most Bound number of solutions on the list Collection. It fails if Bound is anything other than a positive integer.

EXAMPLES

```
?- listing.
% user:likes/2
likes(kev,running).
likes(kev,lifting).
likes(keith,running).
likes(keith,lifting).
likes(ken,swimming).
likes(sally,swimming).
likes(andy,bicycling).
likes(chris,lifting).
likes(chris,running).

yes.
?- setof(Person, likes(Person, Sport), SetOfPeople).
Person = _1
Sport = bicycling
SetOfPeople = [andy];
Person = _1
Sport = lifting
SetOfPeople = [chris,keith,kev];
Person = _1
Sport = running
```

```
SetOfPeople = [chris,keith,kev];

  Person = _1
  Sport = swimming
SetOfPeople = [sally,ken];

no.
?- setof((Sport,People),
        setof(Person, likes(Person, Sport), People),
         Set).
Sport = _1
People = _2
Person = _4
Set =
[(bicycling,[andy]),(lifting,[chris,keith,kev]),

(running,[chris,keith,kev]),(swimming,[sally,ken])]

yes.
?- setof(Person, Sport^(Person likes Sport),
SetOfPeople).
Person = _1
Sport = _2
SetOfPeople = [andy,chris,sally,keith,ken,kev]

yes.
```

SEE ALSO

[Bowen 91, 7.5], [Clocksin 81, 7.8], [Bratko 86, 7.6], [Sterling 86, 17.1].

NAME

`setPrologInterrupt/1`– establish the type of a Prolog interrupt
`getPrologInterrupt/1`– determine the type of a Prolog interrupt

FORMS

```
setPrologInterrupt(Term)
getPrologInterrupt(Term)
```

DESCRIPTION

`Term` is an arbitrary Prolog term.

`setPrologInterrupt(Term)` sets the value of the global interrupt variable to be `Term`.

`getPrologInterrupt(Term)` fetches the value of the global interrupt variable and unifies it with `Term`.

SEE ALSO

`forcePrologInterrupt/1`, `callWithDelayedInterrupt/1`, *User Guide (Prolog Interrupts)*.

NAME

`set_stream_position/2`– seek to a new position in a stream

FORMS

```
set_stream_position(Stream_or_Alias,Position)
```

DESCRIPTION

`set_stream_position/2` is used to change the stream position for a stream which is repositionable.

`Stream_or_Alias` is the stream for which to change the stream position.

`Position` is a term which represents the new position to set. It takes one of the following forms:

- An absolute integer which represents the address of a character in the stream. The beginning of the stream or the first character in the stream has position 0. The second character in the stream has position 1 and so on.
- The atom `beginning_of_stream`.
- The term `beginning_of_stream(N)` where `N` is an integer greater than zero. The position represented by this term is the beginning of the stream plus `N` bytes.
- The atom `end_of_stream`.
- The term `end_of_stream(N)` where `N` is an integer less than or equal to zero. This allows positions (earlier than the end of stream) to be specified relative to the end of the stream. The position represented by this term is the end-of-stream position plus `N` bytes.
- The atom `current_position`.
- The term `current_position(N)` where `N` is an integer. This allows positions to be specified relative to the current position in the file.

EXAMPLES

Suppose that the file "test" is comprised of the characters "abcdefgh\n".

Open and read the first character from the file test.

```
?- open(test,read,_,[alias(test_alias)]),
get_char(test_alias,C).

C = a
```

Seek to two characters before end of file and get the character at this position.

```
?- set_stream_position(test_alias,end_of_stream(-
2)),
?-_  get_char(test_alias,C).

C = h
```

Seek to current position minus two and get the character at this position.

```
?- set_stream_position(test_alias,current_position(-
2)),
?-_  get_char(test_alias,C).

C = g
```

Seek to fourth character in file and get it. Recall that the first character has address 0.

```
?- set_stream_position(test_alias,3),
get_char(test_alias,C).

C = d
```

Get this same character again by backing up one.

```
?- set_stream_position(test_alias,current_position(-
1)),
?-_  get_char(test_alias,C).

C = d
```

Get the next character in the stream.

```
?- get_char(test_alias,C).

C = e
```

Get the current stream position.

```
?- stream_property(test_alias,position(P)).

P = 5
```

Get the current stream position, seek to the beginning of the stream and get that character, then seek back to the old position and get the character at that position.

```
?- stream_property(test_alias, position(P)),
?-_  set_stream_position(test_alias,
beginning_of_stream),
?-_  get_char(test_alias, C1),
?-_  set_stream_position(test_alias, P),
?-_  get_char(test_alias, C2).
P = 5
C1 = a
C2 = f
```

ERRORS

`Stream_or_Alias` is a variable

  ——> `instantiation_error.`

`Position` is a variable

——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias

——> `domain_error(stream_position,Position).`

`Position` is neither a variable nor a stream position

——> `domain_error(stream_position,Position).`

`Stream_or_Alias` is not associated with an open stream

——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` has stream property `reposition(false)`

——> `permission_error(reposition,stream,Stream_or_Alias).`

NOTES

As the example above demonstrates, `set_stream_position/2` may be used when used in conjunction with `stream_property/2`. Typically a program will get the current position using `stream_property/2` and later set the stream position using this saved position.

SEE ALSO

`open/4, get_char/2, stream_property/2,` *User Guide (Prolog I/O).*

NAME

`skip/1` – discard all input characters until specified character

FORMS

```
skip(Char)
```

DESCRIPTION

All characters on the current input stream are discarded up to and including the next character whose ASCII code is `Char`. Skipping past the end-of-file causes `skip/1` to fail.

EXAMPLES

```
?- skip(0’*), get(Next).
Journey To The *s
Next = 115

yes.
```

SEE ALSO

[Clocksin 81, 6.9].

NAME

`sort/2`	– sorts a list of terms
`keysort/2`	– sorts a list of Key-Data pairs

FORMS

```
sort(List,SortedList)
keysort(List,SortedList)
```

DESCRIPTION

`sort/2` sorts the `List` according to the standard order. Identical elements, as defined by `==/2`, are merged, so that each element appears only once in `SortedList`.

`keysort/2` expects `List` to be a list of terms of the form: `Key-Data`. Each pair is sorted by the `Key` alone. Pairs with duplicate `Key`s will not be removed from `SortedList`.

A merge sort is used internally by these predicates at a cost of at most N(log N) where N is the number of elements in `List`.

EXAMPLES

The following examples illustrate the use of `sort/2` and `keysort/2`:

```
?- sort([orange,apple,orange,tangelo,grape],X).
X = [apple,grape,orange,tangelo]
yes.

?- keysort([warren-davidh, bowen-kenneth,
            warren-davids, bowen-david,
            burger-warren], X).
X = [bowen-kenneth,bowen-david,burger-warren,
     warren-davidh,warren-davids]
yes.
```

The following example shows the way structures with the same principal functor are sorted:

```
?- sort([and(a,b,c), and(a,b,a,b), and(a,a),
and(b)],Sorted).
Sorted = [and(a,a),and(a,b,a,b),and(a,b,c),and(b)]
yes.
```

SEE ALSO

`compare/3`, [Bowen 91, 7.4]

NAME

sprintf/3	- formatted write to atoms and strings
bufwrite/2	- formatted write to strings
bufwriteq/2	- formatted write to strings with quoting

FORMS

```
sprintf(Target, Format, Args),
bufwrite(String,Term)
bufwriteq(String,Term)
```

DESCRIPTION

`sprintf/3` is very similar to printf/3, except that instead of writing the formatted out put to a stream, `sprintf/3` writes the output to either a Prolog atom or string. The `Format` and `Args` arguments are identical to the corresponding arguments for `printf/3`. The `Target` argument can be an uninstatiated variable, or can be of one of the forms `atom(A), string(S),` or `list(S)`. In case `Target` is an uninstatiated variable, on success, `Target` is a Prolog double-quoted string containing the formatted output. Similarly, if `Target` is `string(S)` or `list(S)`, on success, `S` is a Prolog double-quoted string containing the formatted output. And if `Target` is `atom(A)`, on success `A` is an atom containing the formatted output.

`bufwrite/2` creates a printed representation of the `Term` using the operator declaration as `write/1` would. However, instead of writing the result to the current output stream, `bufwrite/2` converts the printed representation into a list of ASCII codes. `bufwriteq/2` behaves similarly to `bufwrite/2`, but quotes items exactly the way `writeq/2` would.

EXAMPLES

```
?- sprintf(X, 'Contents: %t, Amount: %t',
[pocket(keys),1]).
X = "Contents: pocket(keys), Amount: 1"

?- sprintf(string(X), 'Contents: %t, Amount: %t',
                      [pocket(keys),1]).
X = "Contents: pocket(keys), Amount: 1"

?- sprintf(atom(X), ’Contents: %t, Amount:
%t’,[pocket(keys),1]).
X = ’Contents: pocket(keys), Amount: 1’

?- bufwrite("jack+f(tom)", +(jack, f(tom))).
yes.
?- bufwrite(S, ’Enterprise’).
S = [69,110,116,101,114,112,114,105,115,101]
yes.
```

SEE ALSO

printf/[2,3,4], write/1, op/3, writeq/1

NAME

`spy/0`	– enable spy points
`spy/1`	– sets a spy point
`nospy/0`	– removes all spy points
`nospy/1`	– removes a spy point

FORMS

```
spy Name/Arity
nospy Name/Arity
nospy
spy(Module:Name/Arity)
nospy(Module:Name/Arity)
```

DESCRIPTION

`spy/1` places a spy point on the procedure name `Name/Arity`. `nospy/1` removes a specific spy point, while `nospy/0` removes all spy points that are currently set. During consult or reconsult, all spy points are disabled, and spy points on predicates which are consulted or reconsulted are removed. `spy/0` re-enables spy points which have been disabled (e.g., those which were not removed, but were disabled during a reconsult).

SEE ALSO

`Tools (Uswing the Debugger), [Bowen 91, 6.5], [Clocksin 81, 6.13], [Bratko 86, 8.4].`

NAME

`statistics/0`	– display memory allocation information
`statistics/2`	– display runtime statistics

FORMS

```
statistics
statistics(runtime,X)
```

DESCRIPTION

`statistics/0` shows the current allocations and amounts used for the working areas of ALS Prolog. `statistics(runtime,X)` unifies X with a two-element list

`[Total,SinceLast]`,

where `Total` is the elapsed cpu time since the start of Prolog execution, and `SinceLast` is the cpu time which has elapsed since the last call to `statistics/2`.

EXAMPLES

```
?- statistics.
Machine State:
   E=ef7fd94 B=efffe1c H=ef7fe44 HB=ef7fe44
TR=efffe1c
   MSP=ef7fd8c SPB=ef7fd94
Clause Space:   55540/262144 (bytes used/total bytes)
yes.

?- statistics(runtime, [Total, SinceLast]).
Total = 1.95
SinceLast = 0.033333333
yes.
```

NOTES

`E,B,H,HB,TR,MSP,SPB` are registers in the Warren Abstract Machine (WAM); cf. [Warren 83], [Warren 86], [Maier 88], Chapter 11.8, [Tick 88].

NAME

`stream_position/3` - reposition a stream

FORMS

```
set_stream_position(Stream_or_alias,
                    Current_Position, New_position)
```

DESCRIPTION

If the stream associated with `Stream_or_alias` supports repositioning, `Current_position` is unified with the current stream position of the stream, and, as a side effect, the stream position of this stream is set to the position represented by `New_position`. `New_position` may be one of the following values:

- An absolute integer which represents the address of a character in the stream. The beginning of the stream or the first character in the stream has position 0. The second character in the stream has position 1 and so on.
- The atom `beginning_of_stream`.
- The term `beginning_of_stream(N)` where `N` is an integer greater than zero. The position represented by this term is the beginning of the stream plus `N` bytes.
- The atom `end_of_stream`.
- The term `end_of_stream(N)` where `N` is an integer less than or equal to zero. This allows positions (earlier than the end of stream) to be specified relative to the end of the stream. The position represented by this term is the end-of-stream position plus `N` bytes.
- The atom `current_position`.
- The term `current_position(N)` where `N` is an integer. This allows positions to be specified relative to the current position in the file.

ERRORS

`Stream_or_Alias` is a variable

——> `instantiation_error.`

`New_Position` is a variable
——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias
——> `domain_error(stream_position,Position).`

`New_Position` is neither a variable nor a stream position
——> `domain_error(stream_position,Position).`

`Stream_or_Alias` is not associated with an open stream
——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` has stream property `reposition(false)`
——>
`permission_error(reposition,stream,Stream_or_Alias)`
`.`

NAME

`stream_property/2` – retrieve streams and their properties

FORMS

`stream_property(Stream,Property)`

DESCRIPTION

`stream_property/2` is used to retrieve information on a particular stream. It may also be used to find those streams satisfying a particular property.

`Stream` may be either an input or output argument. If used as an input argument it should be bound to either a stream descriptor as returned by `open/4` or an alias established in a call to `open/4`. If used as an output argument, `Stream` will only be bound to stream descriptors. This predicate may be used to retrieve those streams whose handles were "lost" for some reason.

`Property` is a term which may take any of the following forms:

> `file_name(F)` – When the stream is connected to a source/sink which is a file, F will be the name of that file.
>
> `stream_name(N)` – `N` is unified with the name of the source/sink regardless of whether the stream is connected to a file or not.
>
> `mode(M)` – M is unified with the I/O mode which was specified at the time the stream was opened.
>
> `input` – The stream is connected to a source.
>
> `output` – The stream is connected to a sink. It is possible for a stream to have both input and output properties.
>
> `alias(A)` – If the stream has an alias, `A` will be unified with that alias.
>
> `position(P)` – If the stream is repositionable, P will be unified with the current position in the stream.

`end_of_stream(E)` – If the stream position is located at the end of the stream, then `E` is unified with '`at`'. If the stream position is past the end of stream, then `E` is unified with '`past`'. Otherwise, `E` is unified with '`no`'. In the current implementation of ALS Prolog, querying about the `end_of_stream` property may cause an I/O operation to result which may block.

`eof_action(A)` – If the stream option `eof_action(Action)` was specified in the options list when the stream was opened, then `A` will be unified with this action. Otherwise, `A` will be unified with the default action appropriate for the stream.

`snr_action(A)` – If the stream option `snr_action(Action)` was specified in the options list when the stream was opened, then `A` will be unified with this action. Otherwise, `A` will be unified with the default action appropriate for the stream.

`reposition(R)` – If positioning is possible on this stream then `R` is unified with `true`; if not `R` is unified with `false`.

`type(T)` – `T` will be unified with either `text` or `binary`, indicating the type of stream.

`maxdepth(D)` – `D` will be unified with the default depth with which terms are written to.

`depth_computation(DC)` – `DC` will be unified with the atom indicating the method of depth computation when writing out terms.

`line_length(L)` – `L` will be unified to the default line length parameter which is used for determining where line breaks should be placed when writing terms.

EXAMPLES

Open 'foo' for write, but "lose" the stream descriptor.

```
?- open(foo,write,_).
```

Use `stream_property` to retrieve the stream descriptor and close it.

```
?- stream_property(S,file_name(foo)), close(S).

S =
stream_descriptor('',closed,file,foo,[noinput|output
],true,

1,0,0,0,0,true,0,wt_opts(78,40000,flat),[],true,text
,
          eof_code,0,0)
```

Open '`foo`' for read with an alias.

```
?- open(foo,read,_,[alias(foo_alias)]).
```

Call stream_property to find out where end-of-stream is. Note that foo was created as the empty file above.

```
?- stream_property(foo_alias,end_of_stream(Where)).

Where = at
```

Call `stream_property` again to find out about the end-of-stream.

```
?- stream_property(foo_alias,end_of_stream(Where)).

Where = past
```

Call stream_property to find out the name of the file associated with the alias.

```
?- stream_property(foo_alias,file_name(Name)).

Name = foo
```

Get all of the names attached to streams.

```
?- setof(N,S^stream_property(S,stream_name(N)),L).

N = N
S = S
L = ['$stderr','$stdin','$stdout',foo]
```

SEE ALSO

`open/4`, `close/1`, `set_stream_position/2`,
`at_end_of_stream/1`,
`set_line_length/2`, *User Guide (Prolog I/O)*.

NAME

`sub_atom/4` – dissect an atom

FORMS

`sub_atom(Atom,Start,Length,SubAtom)`

DESCRIPTION

`sub_atom/4` is used to take apart an atom. The only instantiation requirement is that `Atom` be instantiated to an atom. If `Start` and/or length are instantiated, they must be instantatiated to integers. If `SubAtom` is instantiated, it must be instantiated to an atom.

The `Start` parameter gives the starting character position in `Atom` of the atom `SubAtom`. `Length` is the length of this `SubAtom`. The first character of an atom is considered to begin at position `1`.

`sub_atom/4` is resatisfiable. Upon backtracking all possible values of `Start`, `Length`, and `SubAtom` are generated subject to the initial instantiations of these parameters.

EXAMPLES

```
?- sub_atom('the cat in the hat',5,3,A).

A = cat

yes.
?- sub_atom('the cat in the hat',_,12,A).

A = 'the cat in t';

A = 'he cat in th';

A = 'e cat in the';

A = ' cat in the ';

A = 'cat in the h';
```

```
A = 'at in the ha';

A = 't in the hat';

no.
?- sub_atom('the cat in the hat',S,_,the).

S = 1;
S = 12;

no.
?- sub_atom(in,S,_,A).

S = 1
A = '';

S = 1
A = i;

S = 1
A = in;

S = 2
A = '';

S = 2
A = n;

S = 3
A = '';

no.
```

ERRORS

Atom is a variable

——> `instantiation_error.`

Atom is neither a variable nor an atom

——> `type_error(atom,Atom)`

SubAtom is neither a variable nor an atom

——> `type_error(atom,SubAtom)`

Start is neither a variable nor an integer

——> `type_error(integer,Start)`

Length is neither a variable nor an integer

——> `type_error(integer,Length)`

SEE ALSO

`atom_length/2, atom_concat/3, atom_chars/2, atom_codes/2,`

User Guide (Prolog I/O).

NAME

`system/1`	– Executes the specified OS shell command
`os/0`	– Invokes a subsidiary OS shell (dropped??)

FORMS

```
system(Command)
os
```

DESCRIPTION

`system(Command)` calls the operating system shell with the string `Command` as its argument, where `Command` is either an atom or a string. For example, on Unix,

```
    ?- system('rm myfile.pro').
```

will delete the file *myfile.pro* from the current directory. On those systems for which it is possible, `os/0` invokes a subsidiary operating system shell via a call to `system/1`. When the shell is exited, control is returned to Prolog.

EXAMPLES

```
?- system('pwd').
/usr/elvis/u/chris
yes.

?- system('ls').
RandomNotes  calendar    junkbox    mbox       public
amber        doc         kermrc     papers       test
bin         graphics    mail       prolog      tools
yes.

?- system('alspro').
ALS-Prolog Version 1.0
Copyright (c) 1987, 1988 Applied Logic Systems
?- ^D
```

```
yes.
```

The last ‘`yes`’ was printed by the original ALS Prolog image.

NAME

`tell/1`	– sets the standard output stream
`telling/1`	– returns the name of the standard output stream
`told/0`	– closes the standard output stream

FORMS

```
tell(File)
telling(File)
told
```

DESCRIPTION

`tell/1` sets the current output stream to the file named `File`. If `File` is already open for output, the existing file descriptor will be used. Otherwise, a new file descriptor will be allocated, and output operations will start at the beginning of the file, overwriting the previous contents.

`telling/1` unifies `File` with the name of the current output stream. If no stream has been explicitly opened by `tell`, then `File` will be unified with the atom `user`.

`told/0` closes the current output stream and deallocates its file descriptor. The current output stream is then set to `user`. `user` is the name of the default output stream which is normally connected to the console. `user` is always present, and `told/0` can never close it.

EXAMPLES

The following program will preserve the current output stream, open a file and write one term to it, and then restore the previous output stream.

```
firstTerm(File,Term) :-
        telling(CurrentOutput),
        tell(File),
        write(Term),
        told,
        tell(CurrentOutput).
```

SEE ALSO

`see/1`, `seeing/1`, `seen/0`, User Guide (Prolog I/O),[Bowen 91, 7.8], [Clocksin 81, 5.4], [Bratko 86, 6.1].

NAME

`term_chars/2`	– convert between a term and the list of characters which represent the term
`term_codes/2`	– convert between a term and the list of character codes which represent the term

FORMS

```
term_chars(Term,CharList)
term_codes(Term,CodeList)
```

DESCRIPTION

If `CharList` is bound to a list of characters then it is parsed according to the syntax rules for terms. Should the parse be successful, the resulting value is unified with `Term` in a call to `term_chars/2`.

If `CodeList` is bound to a list of character codes then it is is parsed according to the syntax rules for terms. Should the parse be successful, the resulting value is unified with `Term` in a call to `term_codes/2`.

Otherwise `CharList` (or `CodeList`) will be bound to the list of characters (character codes) which would result as output from `write_canonical(Term)`

EXAMPLES

```
?- term_chars(p(a,X), L).

X = X
L = [p,'(',a,',','_','A',')']

yes.
?- term_codes(A,"X = /* a comment */ 3+4").

A = (_A = 3+4)
```

```
yes.
?- term_codes(A,"foo bar").

Error: Syntax error.
- Goal:          builtins:term_codes(_A,"foo bar")
- Error Attribute: syntax('foo barend_of_file\n   ^',
                    'Fullstop (period) expected',1,
                 stream_descriptor('',closed,string,
                         string("foo
bar"),[input|nooutput],false,3,

[],0,0,0,true,1,wt_opts(78,40000,flat),[],
                         true,text,eof_code,0,0))
- Throw pattern: error(syntax_error,
                     [builtins:term_codes(_A,*),
                     syntax('foo barend_of_file\n
^',
                             'Fullstop (period)
expected',1,

stream_descriptor('',closed,string,*,*,

false,3,[],0,0,0,true,1,*,[],true,
                             text,eof_code,0,0))])
```

ERRORS

`CharList` is bound to a list, but the list does not contain characters

——> `domain_error(character_list,CharList)`

`CodeList` is bound to a list, but the list does not contain character codes

——> `domain_error(character_code_list,CodeList)`

`CharList` (or `CodeList`) is not parsable as a term

——> `syntax_error`

SEE ALSO

read_term/3, write_canonical/2, number_chars, atom_chars,
User Guide (Prolog I/O).

NAME

`time/1`	– gets the current system time
`date/1`	– gets current date

FORMS

```
time(Time)
date(Date)
```

EXAMPLES

```
?- time(Time).

Time = (16:51:49)
yes.
?- date(Date)
Date = 93/11/5

yes.
```

NAME

`trace/0`	– turn on tracing
`trace/1`	– trace the execution of a goal
`notrace/0`	– turn off tracing

FORMS

```
trace Goal
trace(Goal)
trace
notrace
```

DESCRIPTION

In the `trace/1` case, the `Goal` will be single stepped according to the debugger's leash mode. In the `trace/0` case, tracing will be turned on for all items until a call to `notrace/0` is encountered.

EXAMPLES

```
?- trace(append([a,b,c],[d],X)).
(1) 1 call: append([a,b,c],[d],_11) ?
(2) 2 call: append([b,c],[d],_94) ?
(3) 3 call: append([c],[d],_170) ?
(4) 4 call: append([],[d],_246) ?
(4) 4 exit: append([],[d],[d]) ?
(3) 3 exit: append([c],[d],[c,d]) ?
(2) 2 exit: append([b,c],[d],[b,c,d]) ?
(1) 1 exit: append([a,b,c],[d],[a,b,c,d]) ?
```

SEE ALSO

`spy/1`, `nospy/0`, `leash/0`,

[Bowen 91, 4.5], [Bratko 86, 8.4], [Clocksin 81, 8.3].

NAME

`true/0` – always succeeds

FORMS

```
true
```

DESCRIPTION

`true/0` succeeds once, and fails upon backtracking.

EXAMPLES

```
?- true.
yes.
```

SEE ALSO

`fail/0`,

[Clocksin 81, 6.2].

NAME

`ttyflush/0` – forces all buffered output to the screen

FORMS

```
ttyflush
```

DESCRIPTION

`ttyflush/0` is used to make sure that output from tty I/O predicates appears on the screen when you want it to. Screen output is normally flushed whenever Prolog is waiting on some I/O operation. Typically, the I/O operation that is waited on is some form of `tty` read. In this case, you don't have to use `ttyflush`.

A long computation, while causing you to wait for a response, does not qualify as an I/O operation, so any output predicates that were run before the CPU hog started, might not show their output on the screen until the long computation is finished. To combat this problem, `ttyflush/0` can be used before entering into the long computation section of your program.

EXAMPLES

If we define the following useless predicate:

```
infiniteLoop :- infiniteLoop.
```

and then try to write something to the screen before running it:

```
?- printf("April Fool’s Day"), infiniteLoop.
April Fool's Day    Break Handler
    ------------------------
    a - Abort Computation
    b - Break shell
    c - Continue
    d - Debug
    e - Exit Prolog
    f - Fail
    p - Return to Previous Break Level
    s - Show goal broken at
    t - Stack trace
    ? - This message
Break(1) >a

Warning: Aborting from Control-C or Control-Break.

Error: Execution aborted.
```

we find that the output doesn't appear until the `Control-C` is pressed. We can avoid this problem by putting a call to `ttyflush/0` after the message printing predicate. The following example shows the result:

```
?- printf("April Fool’s Day"), ttyflush,
infiniteLoop.
April Fool's Day    Break Handler
    ------------------------
    a - Abort Computation
    b - Break shell
    c - Continue
    d - Debug
    e - Exit Prolog
    f - Fail
    p - Return to Previous Break Level
    s - Show goal broken at
    t - Stack trace
    ? - This message
Break(1) >a

Warning: Aborting from Control-C or Control-Break.

Error: Execution aborted.
```

NOTES

The ISO Standard mandates that `flush_output/[0,1]` is preferred over `ttyflush/0`.

SEE ALSO

printf/1, write/1, put/1, flush_output/[0,1].

NAME

`'$uia_alloc'/2`	–	allocates a UIA of specified length
`'$uia_size'/2`	–	obtains the actual size of a UIA
`'$uia_clip'/2`	–	clip the given UIA
`'$uia_pokeb'/3`	–	modifies the specified byte of a UIA
`'$uia_peekb'/3`	–	returns the specified byte of a UIA
`'$uia_pokew'/3`	–	modifies the specified word of a UIA
`'$uia_peekw'/3`	–	returns the specified word of a UIA
`'$uia_pokel'/3`	–	modifies the specified long word of a UIA
`'$uia_peekl'/3`	–	returns the specified long word of a UIA
`'$uia_poked'/3`	–	modifies the specified double of a UIA
`'$uia_peekd'/3`	–	returns the specified double of a UIA
`'$uia_pokes'/3`	–	modifies the specified substring of a UIA
`'$uia_peeks'/3`	–	returns the specified substring of a UIA
`'$uia_peeks'/4`	–	returns the specified substring of a UIA
`'$uia_peek'/4`	–	returns the specified region of a UIA
`'$uia_poke'/4`	–	modifies the specified region of a UIA
`'$strlen'/2`	–	returns the length of the specified symbol

FORMS

```
'$uia_alloc'(BufLen,UIABuf)
'$uia_size'(UIABuf,Size)
'$uia_clip'(UIABuf,Size)
'$uia_pokeb'(UIABuf,Offset,Value)
'$uia_peekb'(UIABuf,Offset,Value)
'$uia_pokew'(UIABuf,Offset,Value)
'$uia_peekw'(UIABuf,Offset,Value)
'$uia_pokel'(UIABuf,Offset,Value)
'$uia_peekl'(UIABuf,Offset,Value)
'$uia_poked'(UIABuf,Offset,Value)
'$uia_peekd'(UIABuf,Offset,Value)
'$uia_pokes'(UIABuf,Offset,Symbol)
'$uia_peeks'(UIABuf,Offset,Symbol)
```

```
'$uia_peeks'(UIABuf,Offset,Size,Symbol)
'$uia_peek'(UIABuf,Offset,Size,Value)
'$uia_poke'(UIABuf,Offset,Size,Value)
'$strlen'(Symbol,Size)
```

DESCRIPTION

A call to `'$uia_alloc'(BufLen,UIABuf)`creates a UIA of the length specified by `BufLen`. `BufLen` should be instantiated to a positive integer which represents the size (in bytes) of the UIA to allocate; currently the maximum allowable value of `BufLen` is 1024. The actual size of the buffer allocated will be that multiple of four between `BufLen+1` and `BufLen+4`. (UIAs are allocated on word boundaries and an extra byte is added to provide for zero termination of strings when UIAs are used for symbols.) `UIABuf` should be a variable. UIAs are initially filled with zeros, and will unify with the null atom (`''`).

The call `'$uia_size'(UIABuf,Size)`returns the actual size (in bytes) of the given UIA. If `Size` is less than or equal to the actual size of the given `UIABuf`, the call `'$uia_clip'(UIABuf,Size)` reduces the size of `UIABuf` by removing all but one of the trailing zeros (null bytes).

Single-byte values can be inserted into a UIA buffer using `'$uia_pokeb'/3`. The modifications are destructive, and do not disappear upon backtracking. These procedures can be used to modify system atoms (file names and strings that are represented as UIAs). However, this use is strongly discouraged. `UIABuf` should be a buffer obtained from `'$uia_alloc'/2`. `Offset` is the offset within the buffer to the place where `Value` is to be inserted. Both `Offset` and `Value` are integers. In `'$uia_pokeb'/3`, the buffer is viewed as a vector of bytes with the first byte having offset zero. The byte at `Offset` from the beginning of the buffer is changed to `Value`. The companion predicates `'$uia_pokew'/3`, `'$uia_pokel'/3`, `'$uia_poked'/3`, perform the corresponding operation on words, long words, and doubles, respectively.

`'$uia_peekb'/3` is used to obtain specific bytes from a UIA buffer created by `'$uia_alloc'/2`, or from any other UIA existing in the system. The parameters for these procedures are specified as follows: The arguments of

`'$uia_peekb'/3` are interpreted in the same manner as the parameters for `'$uia_pokeb'/3`. The parameter `Symbol` must be a UIA or an atom. The parameter `Size` must be an integer. The companion predicates, `'$uia_peekw'/3, '$uia_peekl'/3, '$uia_peekd'/3,` perform the corresponding operation on words, long words, and doubles, respectively.

Like `'$uia_pokeb'/3`, `'$uia_pokes'/3` views the buffer as a vector of bytes with offset zero specifying the first byte. But instead of replacing just a single byte, `'$uia_pokes'/3` replaces the portion of the buffer beginning at `Offset` and having length equal to the length of `Symbol`, using the characters of `Symbol` for the replacement. If `Symbol` would extend beyond the end of the buffer, `Symbol` is truncated at the end of the buffer. The parameter `Symbol` must be an atom. The parameter `Size` must be an integer.

`'$uia_peeks'/3` binds `Symbol` to a UIA consisting of the characters beginning at position `Offset` and extending to the end of the buffer. `'$uia_peeks'/4` binds `Symbol` to a UIA consisting of the characters beginning at position `Offset` and extending to position `End` where `End = Offset + Size`. If End would occur beyond the end of the buffer, `Symbol` simply extends to the end of the buffer.

Provided that `Offset` and `Size` define a proper region within the given `UIABuf` (i.e., not including the final byte of `UIABuf`), `'$uia_poke'(UIABuf,Offset,Size,Value)` modifies the indicated region by copying characters from the given UIA (or symbol) `Value`. The size of the atom or UIA `Value` must be greater than or equal to `Size`. The region copied from ‘Value’ is defined by offset `0` and `Size`.

Provided that `Offset` and `Size` define a proper region within the given `UIABuf` (i.e., not including the final byte of `UIABuf`), `'$uia_peek'(UIABuf,Offset,Size,Value)` extracts the indicated region from `UIABuf`, returning it as a new UIA `Value`.

When `Symbol` is a Prolog symbol (atom or UIA), `'$strlen'(Symbol,Size)` returns the length of the print name of that symbol (thus not counting the terminating null byte).

EXAMPLES

```
copy_atom_to_uia(Atom, UIABuf) :-
       name(Atom,ExplodedAtom),
       copy_list_to_uia(ExplodedAtom,UIABuf).

copy_list_to_uia(Ints,UIABuf) :-
       length([_|Ints], BufLen),
       '$uia_alloc'(BufLen, UIABuf),
       copy_list_to_uia(Ints, 0, UIABuf).

copy_list_to_uia([],_,_) :- !.
copy_list_to_uia([H | T], N, Buf) :-
       '$uia_pokeb'(Buf,N,H),
       NN is N+1,
       copy_list_to_uia(T, NN, Buf).
```

SEE ALSO

atom_concat/3, sub_atom/3, atom_char/2

NAME

`var/1`	– the variable is unbound
`nonvar/1`	– the variable is instantiated

FORMS

```
var(Term)
nonvar(Term)
```

DESCRIPTION

`var/1` succeeds if `Term` is an unbound variable, and fails otherwise.

`nonvar/1` succeeds when `Term` is a constant or structured term.

EXAMPLES

```
?- var(constant).

no.
?- nonvar(constant).

yes.
?- X=Y, Y=Z, Z=doughnut, var(X).
no.
```

SEE ALSO

[Bowen 91, 7.6], [Bratko 86, 7.1.1], [Sterling 86, 10.1].

NAME

`write/1`	– write term to current output stream
`write/2`	– write term to specified stream
`writeq/1`	– write term to current output stream so that it may be read back in
writeq/2	– write term to specified stream so that it may be read back in
`write_canonical/1`	– write term to current output stream in canonical form (no operators)
`write_canonical/2`	– write term to specified stream in canonical form
`write_term/2`	– write term to current output stream with options
`write_term/3`	– write term to specified output stream with options
display/1	– write term to current output stream in canonical form

FORMS

```
write(Term)
write(Stream_or_Alias,Term)
writeq(Term)
writeq(Stream_or_Alias,Term)
write_canonical(Term)
write_canonical(Stream_or_Alias,Term)
write_term(Term,Options)
write_term(Stream_or_Alias,Term,Options)
display(Term)
```

DESCRIPTION

These predicates will output the term bound to Term to a stream. The format of the term is controlled by which variant is called or by an option given to `write_term`. None of these procedures output a fullstop after the term written.

`write/1` behaves as if `write/2` were called with the current output stream as the `Stream_or_Alias` argument.

`write/2` behaves as if `write_term/3` were called with `Options` bound to

```
[quoted(false),numbervars(true),lettervars(false),
     line_length(1024).
```

In addition, the default line length is ignored and set to a large number, causing the output of long terms to not be pretty printed. Variables are printed as underscore followed by some number.

`writeq/1` behaves as if `writeq/2` were called with the current output stream as the `Stream_or_Alias` argument.

`writeq/2` behaves as if `write_term/3` were called with `Options` bound to

```
[quoted(true),numbervars(true),lettervars(true)].
```

The line length is set to the default line length for the stream which is being output to. Variables are printed as an underscore followed by a capital letter. `writeq` is useful for outputting a term whichmight be later subject to a read from Prolog.

`write_canonical/1` behaves as if `write_canonical/2` were called with the current output stream bound to the `Stream_or_Alias` argument.

`write_canonical/2` behaves as if `write_term/3` were called with `Options` bound to

```
[quoted(true),ignore_ops(true),lettervars(true)].
```

This is the same behavior supplied by the DEC-10 compatiblity predicate `display/1`. `write_canonical` is useful in situations where it is desirable to output a term in a format which may subsequently read in without regard to operator definitions. Such terms are not particularly pleasing to look at, however.

`write_term/2` behaves as if `write_term/3` were called with the current output stream bound to Stream_or_Alias.

`write_term/3` writes out the term `Term` to the output stream associated with `Stream_or_Alias` and subject to the options in the write option list `Options`. The options in the write options list control how a term is output.

The options mandated by the draft standard are:

quoted(Bool) – Bool is true or false. When Bool is true, atoms and functors are written out in such a manner so that read/[1,2] may be used to read them back in; when Bool is false indicates that symbols should be written out without any special quoting; control characters embedded in an atom will be written out as is.

ignore_ops(Bool) – Bool may be true or false. When Bool is true, compound terms are output in functional notation.

numbervars(Bool) – When Bool is true, a term of the form '$VAR'(N), where N is a non-negative integer, will be output as a variable name consisting of a capital letter possibly followed by an integer. The capital letter is the (i+1)th letter of the alphabet, and the integer is j, where i = N mod 26 and j = N div 26. The integer j is omitted if it is zero.

Other options not mandated by the draft standard but supported by ALS Prolog are:

lettervars(Bool) – If Bool is true, variables will be printed out as an underscore followed by a letter and digits if necessary. If Bool is false, variables will be printed as _N, where N is computed using the address where the variable lives at. This latter mode is more suited to debugging purposes where correspondences between variables in various calls is required.

maxdepth(N,Atom1,Atom2) – N is the maximum depth to print to. Atom1 is the atom to output when this depth has been reached. Atom2 is the atom to output when this depth has been reached at the tail of a list.

maxdepth(N) – same as maxdepth(N,*,...)

depth_computation(Val) – Val may be either flat or nonflat. This indicates the method of depth computation. If Val is bound to flat, all arguments of a term or list will be treated as being at the same depth. If Val is nonflat, then each subsequent argument in a term (or each subsequent element of a list) will be considered to be at a depth one greater than the preceding structure argument (or list element).

line_length(N) – N is the length in characters of the output line. The pretty printer will attempt to break lines before they exceed the given line length.

indent(N) – N specifies the initial indentation in characters to use for the second and subsequent lines output (if any).

`quoted_strings(Bool)` – If `Bool` is `true`, lists of suitable character codes will print out as double quoted strings. If `false`, these lists will print out as lists of small integers.

EXAMPLES

```
?- X = 'Hello\tthere',
?-_  write(X),nl,
?-_  writeq(X),nl,
?-_  write_canonical(X),nl.
Hello    there
'Hello\tthere'
'Hello\tthere'

X = 'Hello\tthere'


?- T = [3+4,'$VAR'(26) * X - 'Y'],
?-_  write(T), nl,
?-_  writeq(T), nl,
?-_  write_canonical(T), nl.
[3+4,A1*_4100-Y]
[3+4,A1*_A-'Y']
.(+(3,4),.(-(*('$VAR'(26),_A),'Y'),[]))

T = [3+4,'$VAR'(26)*X-'Y']
X = X



?- L =
[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
],
?-_  write_term(L,[line_length(20)]), nl,
?-_  write('list: '),
?-_ write_term(L,[line_length(26),indent(6)]),nl.
```

```
[a,b,c,d,e,f,g,h,i,
      j,k,l,m,n,o,p,
      q,r,s,t,u,v,w,
      x,y,z]
list: [a,b,c,d,e,f,g,h,i,
             j,k,l,m,n,o,p,
             q,r,s,t,u,v,w,
             x,y,z]

L =
[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
]


?- S = "A string",
?-_  write_term(S, [quoted_strings(true)]),nl,
?-_  write_term(S, [quoted_strings(false)]),nl.
"A string"
[65,32,115,116,114,105,110,103]

S = "A string"


?- T = [a(b(c(d))), a(b(c(d))), a(b(c(d))),
a(b(c(d)))],
?-_  write_term(T, [maxdepth(3),
depth_computation(flat)]), nl,
?-_  write_term(T, [maxdepth(3),
depth_computation(nonflat)]), nl.
[a(b(*)),a(b(*)),a(b(*)),a(b(*))]
[a(b(*)),a(*),*,...]

T = [a(b(c(d))),a(b(c(d))),a(b(c(d))),a(b(c(d)))]
```

ERRORS

`Stream_or_Alias` is a variable
  ——> `instantiation_error.`

`Stream_or_Alias` is neither a variable nor a stream descriptor nor an alias
——> `domain_error(stream_or_alias,Stream_or_Alias).`

`Stream_or_Alias` is not associated with an open stream
——> `existence_error(stream,Stream_or_Alias).`

`Stream_or_Alias` is not an output stream
——> `permission_error(output,stream,Stream_or_Alias).`

Options is a variable
——> `instantiation_error.`

Options is neither a variable nor a list
——> `type_error(list,Option).`

Options is a list an element of which is a variable
——> `instantiation_error.`

Options is a list containing an element E which is neither a variable nor a valid write option
——> `domain_error(write_option,E).`

SEE ALSO

read_term/[2,3], read/[1,2], open/4, close/1, nl/[0,1], put_char/[1,2], put_code/[1,2], set_line_length/2, op/3, tell/1,*User Guide (Prolog I/O), [Bowen 91, 7.8], [Clocksin 81, 6.9], [Sterling 86, 12.1], [Bratko 86, 6.2.1].*

31 ASCII Table

000 nul	032 sp	064 @	096 ‘
001 soh	033 !	065 A	097 a
002 stx	034 "	066 B	098 b
003 etx	035 #	067 C	099 c
004 eot	036 $	068 D	100 d
005 enq	037 %	069 E	101 e
006 ack	038 &	070 F	102 f
007 bel	039 ’	071 G	103 g
008 bs	040 (	072 H	104 h
009 ht	041)	073 I	105 i
010 nl	042 *	074 J	106 j
011 vt	043 +	075 K	107 k
012 np	044 ,	076 L	108 l
013 cr	045 -	077 M	109 m
014 so	046 .	078 N	110 n
015 si	047 /	079 O	111 o
016 dle	048 0	080 P	112 p
017 dc1	049 1	081 Q	113 q
018 dc2	050 2	082 R	114 r
019 dc3	051 3	083 S	115 s
020 dc4	052 4	084 T	116 t
021 nak	053 5	085 U	117 u
022 syn	054 6	086 V	118 v
023 etb	055 7	087 W	119 w
024 can	056 8	088 X	120 x
025 em	057 9	089 Y	121 y
026 sub	058 :	090 Z	122 z

027 esc	059 ;	091 [	123 {
028 fs	060 <	092 \	124 |
029 gs	061 =	093]	125 }
030 rs	062 >	094 ^	126
031 us	063 ?	095 _	127 del

Table 20. ASCII Character Set (Decimal)