Flex & Prolog Utility Predicates

VisiRule combines graphical modelling with AI programming.

VisiRule generates flex code which in turn compiles into Prolog.

Each of these sub-systems offer a wide range of routines for manipulating text, numbers, inputs, outputs etc.

You can think of these as scripting languages – though they are actually compiled on-the-fly

Theis document seeks to show how to programmatically interact with certain constructs such as global variables, questions, groups from within VisiRule code boxes and statements boxes.

Flex

Global Variables
Flex supports global variables which can be accessed from anywhere within a chart. The routines which are relevant are add-value, isa_value and new_value.

Questions
Flex supports program access to questions thru isa_question and new_question. The contents of questions are accessible as global variables reflecting the name of the question.

Groups
Groups are arbitrary collections of items. The routines which are relevant are isa_group and new_group.

add_value(+Slot,+Term)

`add_value/2` can either be used to augment lists or to increment numbers.

The given *`Term`* is first dereferenced down to some *`Value`*.
The *`Slot`* should be the name of a global variable.

If there is an existing slot then it is updated with *`Value`*. Otherwise, a new slot is created.

Example

```
add_value( colors, [red,blue,cyan,white] ).
```

isa_group(?Name,-Elements)

Retrieve the *`Name`* and *`Elements`* of a group.

Example

```
?- isa_group( N, E) .
N = colors
E = [black,blue,green,cyan,red,magenta,yellow,white])
```

isa_question(?Name,-Question,-Answer,-Explanation)

Retrieve a question from the workspace (see the `new_question/4` predicate for a description of the arguments).

Example

```
?- isa_question( N, Q, A, E ).
N = name_of_applicant
Q = ['Please',enter,your,full,name]
A = input(name)
E = text(['No',name,means,no,benefit,!])
N = starter
Q = ['Please',choose,a,starter,for,your,meal]
A = single([pate,soup,melon])
E = none
```

isa_value(?Slot,-Value)

Retrieve either the current *`Value`*, or in its absence the default *`Value`*, for the *`Slot`*. The *`Slot`* should be the name of a global variable.

Examples

```
?- isa_value( temperature, X ).
X = 100
```

new_group(+Name,+Elements)

Create (or replace if the *Name* already exists) a new group containing the given *Elements*.
A group is used as an ordering relation when comparing the relative values of two elements (see comparison/4).

Examples

```
new_group( fuzzy_ordering,
[impossible,improbable,possible,probable,definite])
```

new_question(+Name,+Question,+Answer,+Explanation)

Add (or replace if the *Name* already exists) a new question to the workspace.
Question is a list of words to be displayed whenever the question is asked
The *Answer* indicates how an answer is to be obtained. It is one of the following:

- `input` Set up a dialog with an edit field into which the user can type words and numbers.
- `input(`*T*`)` Set up a dialog with an edit field into which the user can type information. The expected type of the input is determined by *T*, which is one of `set`, `name`, `number`, `integer` or `( X : `*conditions*` )`
- `single(`*M*`)` Set up a dialog with a menu *M* from which the user can make a single selection.
- `multiple(`*M*`)` Set up a dialog with a menu *M* from which the user can make multiple selections.
- *Term* `:` *Goal* Execute the *Goal* and return the *Term* as the answer to the question.

Examples

```
new_question( name_of_applicant,
     ['Please',enter,your,full,name],
     input(name),
     text(['No',name,means,no,benefit,!]) )

new_question( starter,
     ['Please',choose,a,starter,for,your,meal],
     single([pate,soup,melon]), none )

new_question( dessert,
     ['Please',choose,a,dessert,for,your,meal],
     multiple(dessert),
     file(the_complete_irish_cook) )
```

new_value(+Slot,+Term)

The given *Term* is first dereferenced down to some *Value*.
If the *Slot* is the name of a global variable then the following call is made:
`new_slot( Slot, global, Value) .`
The frame `global` is a special frame reserved for the value of global variables.
Example

```
new_value( temperature, boiling_point) .
```

prove(+Goal)

Dereference each of the arguments of the *Goal* and then try to prove it.
If there is a Prolog program for the *Goal*, then that program is run, otherwise the workspace is searched for a matching fact.

reconsult_rules(+FileName)

Reconsult (i.e. load) a flex ksl file.

restart

Clear the workspace of all instances, slot values, all facts and all exceptions, re-enable all rules and finally run all data directives. It is defined by the program:

```
restart :-
     remove_instances,
     remove_slots,
     remove_facts,
     remove_exceptions,
     enable_rules,
     run_data.
```

Example Uses

These are actual extracts from the Console window command line.

Here we use add_value to update a global set

```
| ?- new_value( colors, [black,white] ).
yes

| ?- prove( write( colors ) ), nl.
[black,white]
yes

| ?- add_value( colors, [red,blue,cyan,white]  ).
yes

| ?- prove( write( colors ) ), nl.
[red,blue,cyan,black,white]
yes
```

Note:

1] When using the Prolog Console Window and command line, we have to explicitly wrap up the call to write inside a prove/1 to invoke dereferencing.

2] add_value knows not to duplicate entries in lists denoted by [] as they are sets in Flex.

Here we use add_value to update a global counter

```
| ?- new_value( color_count, 2  ).
yes

| ?- prove( write( color_count ) ), nl.
2
yes

| ?- add_value( color_count, 3  ).
yes

| ?- prove( write( color_count ) ), nl.
5
Yes
```

Prolog level predicates

Prolog is a powerful AI programming language with many many predicates (routines).

We introduce some of the ones here which are particularly useful when building VisiRule solutions.

---

is/2

compute an arithmetic expression

Result is Expression
?Result <number> or <variable>
+Expression <expr>

Comments This evaluates the given arithmetic Expression and unifies the solution with Result. The Expression may be anything from a simple number to a deeply nested term containing one or more of the subterms, representing functions and operators. These can be split into two groups, the first of which can be applied to any numbers (integer or floating point) and the second of which are limited to use on integers. The following table shows the general functions and operators:

Term	**Function**
X + Y	adds X to Y
X - Y	subtracts Y from X
- X	returns the negative of X
X * Y	multiplies X by Y
X / Y	divides X by Y
X // Y	performs integer division of X by Y, truncating the result towards zero
X mod Y	computes X modulo Y, where the result has the same sign as Y
X ^ Y	raises X to the power of Y
?(X)	computes a linear congruential pseudo random floating point number between zero and X

@(X)	computes a Marsaglia Zaman pseudo random floating point number between zero and X
abs(X)	computes the absolute value of X
acos(X)	computes the arccosine of X (degrees)
aln(X)	computes the natural antilogarithm of X
alog(X)	computes the common antilogarithm of X
asin(X)	computes the arcsine of X (degrees)
atan(X)	computes the arctangent of X (degrees)
cos(X)	computes the cosine of X (degrees)

Examples The following calls show a variety of expressions being evaluated or tested:

?- **X is 2 + 2.** <enter>
X = 4
?- **5 is 10 / 2.** <enter>
yes
?- **X is 22 / 7.** <enter>
X = 3.142857142857143
?- **X is asin(sin(45) * cos(60) / tan(75)).** <enter>
X = 5.436029972077099

Notes The is/2 predicate is the primary "maths engine" in Prolog programs, being used for everything from the simple incrementing of a counter through to complex arithmetical computations. In addition to the basic set of standard arithmetic operators, **WIN-PROLOG** includes a full set of "scientific calculator" functions, covering logarithms, linear trigonometry, truncation and rounding functions. For simulation work it also includes a carefully-researched pseudo random number generator (see seed/1 for further information), and finally a set of bit-oriented integer manipulation operators.

There are various things to note about arithmetic handling in **WIN-PROLOG**: firstly, there is automatic conversion between the integer and floating point data types during computations, and wherever possible, results are converted back to integers upon completion. Consider the following call:

?- **X is 2 * 4.5.** <enter>
X = 9

The is/2 predicate needs to multiply the integer "2" with the floating point number "4.5", and so it converts the former into the floating point value "2.0" prior to the multiplication. Before

returning the result "9.0", a check is made to see whether this value can be represented precisely by an integer; in this case it can, so the computed result is converted back into the integer "9", and it is this that is returned.

This new package is faster and more accurate than its predecessor, allowing an extra digit or so of usable precision, which is reflected in output, which now defaults to 16 significant digits, rather than the 15 of earlier versions of

From WIN-PROLOG.WIN-PROLOG 6.0 - Technical Reference 379

---

length/2

test or get the length of a list or generate a list

length(List, Length)
?List <list> or <variable>
?Length <integer> or <variable>

Comments This can be used to get ot check the length of a list, or to generate lists of a given length. When Length is an integer, a list of distinct variables is generated and unified with List. When Length is an unbound variable, the length of List is computed and unified with Length. This predicate is non-deterministic, and can generate successive lengths and lists on backtracking.

Examples When called with two variables, a succession of lists and lengths is generated:

?- **length(L, N).** <enter>
L = [] ,
N = 0 ; <space>
L = [_1] ,
N = 1 ; <space>
L = [_1,_2] ,
N = 2 <enter>

Notes The length/2 predicate can be used to test and generate lists, and because it is written in Prolog, it can also handle non-deterministic cases as shown above.

---

member/3

get or check a member of a list and its position

member(Term, List, Position)
?Term <term>
?List <list> or <variable>

?Position <integer> or <variable>

Comments This predicate succeeds when its first argument is bound to a Term which is a member of the List bound to the second argument, and its third argument is bound to this element's Position. The second argument may be a fully or partially instantiated list, or simply a variable; member/3 can backtrack to generate alternative solutions where appropriate.

Examples The following command extracts each of the elements of a given list in turn:

?- **member(T, [black,and,white], P).** <enter>
T = black ,
P = 1 ; <space>
T = and ,
P = 2 ; <space>
T = white ,
P = 3 ; <space>
no

Notes The member/3 predicate is a special variant of the classic Prolog program, member/2: when it is called with Position as an unbound variable, the present predicate is exactly like its arity-two sibling, supporting all the latter's backtracking and list-generating features. When, however, Position is given as an integer, member/3 becomes deterministic, returning just the given solution and leaving no choicepoints.

---

sort/2

sort a list into ascending order, removing duplicate terms

sort(List1, List2)
+List1 <list>
?List2 <variable> or <list>

Comments This sorts the list of terms in List1 into ascending order according to the standard ordering of terms, removes all duplicates, and unifies the result with List2.

Examples The following command sorts the given list of terms into ascending order, removing any duplicates:

?- **sort([the,cat,and,the,dog], S).** <enter>
S = [and,cat,dog,the]

---

sort/3

sort a list into ascending order using given key path

sort(List1, List2, Path)
+List1 <list>
-List2 <variable>
+Path <list>

Comments This sorts the list of terms in List1 into ascending order according to the "standard ordering" of terms, using the given Path to identify the sort key, and unifies the result with List2.

Examples The following command sorts the given list of terms into ascending order, using an empty path ("[]"), and without removing any duplicates:

?- **sort([the,cat,and,the,dog], S, []).** <enter>
S = [and,cat,dog,the,the]

Notes Unlike sort/2, the sort/3 predicate does not remove duplicate entries from the sorted list, and unlike keysort/2, it does not restrict the input list to containing only terms of the form "key-value". Of the three sorting predicates, sort/3 is the most general: in fact, the other two are simply implemented in terms of the present predicate.

The Path feature of sort/3 is extremely powerful, allowing any arbitrary sub-term to be used as the sort key on any particular occasion (see mem/3 for more information about paths):

?- **sort([3-the,1-quick,4-brown,2-fox], S, [2]).** <enter>
S = [1 - quick,2 - fox,3 - the,4 - brown]
?- **sort([3-the,1-quick,4-brown,2-fox], S, [3]).** <enter>
S = [4 - brown,2 - fox,1 - quick,3 - the]

---

remove/3

remove an element from a list

remove(Term, List, Rest)
?Term <term>
?List <list> or <variable>
?Rest <list> or <variable>

Comments This predicate succeeds when its first argument is bound to a Term that is an element in a particular List, and Rest is bound to another list that contains all the elements of List except for Term. Either of the last two arguments may be fully or partially

instantiated lists, or simply variables; remove/3 can backtrack to generate alternative solutions where appropriate.

Examples The following command simply removes the element, "2", from a list, "[1,2,3]", to give the remainder:

?- **remove(2, [1,2,3], R).** <enter>
R = [1,3] ; <space>
no

The next example runs this predicate in reverse, inserting the given element into a list; on backtracking, each possible solution is offered in turn:

?- **remove(a, L, [1,2]).** <enter>
L = [a,1,2] ; <space>
L = [1,a,2] ; <space>
L = [1,2,a] ; <space>
no

Notes The remove/3 predicate is a classic Prolog program, and is widely used for removing items from or inserting them into lists: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of remove/3. In order to avoid errors, such references must be renamed or removed before loading files into **WIN-PROLOG**.

---

reverse/2

reverse the order of elements in a list

reverse(List, Reverse)
?List <list> or <variable>
?Reverse <list> or <variable>

Comments This predicate succeeds when both its first argument is bound to a List, and its second argument contains a list comprised of the same set of elements, but in Reverse order. Either of the two arguments may be fully or partially instantiated lists, or simply variables; reverse/2 can backtrack to generate alternative solutions where appropriate.

Examples The following command simply reverses the list, "[1,2,3]", to give a second list:

?- **reverse([1,2,3], R).** <enter>
R = [3,2,1]

Notes The reverse/2 predicate is a classic Prolog program, and is widely used for reversing the order of elements in lists: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of reverse/2. In order to avoid errors, such references must be renamed or removed before loading files into **WIN-PROLOG**.

When its first argument is given as a list, reverse/2 is very efficient (its behaviour is linear with respect to list length), because it uses a "difference list" algorithm; however, where the first argument is a variable, the algorithm reverts to "generate and test", which explains the eventual "heap full" error on backtracking. For this reason, it is always best to specify the first argument in calls to reverse/2.

---

append/3

join or split arbitrary lists

append(First, Second, Whole)
?First <list> or <variable>
?Second <list> or <variable>
?Whole <list> or <variable>

Comments This predicates succeeds when Whole is bound to a list consisting of the Second list appended to the First list. Any of the arguments may be fully or partially instantiated lists, or simply variables; append/3 can backtrack to generate alternative solutions where appropriate.

Examples The following command simply joins two list, "[1,2,3]" and "[a,b,c]", to give a new whole one:

?- **append([1,2,3], [a,b,c], W).** <enter>
W = [1,2,3,a,b,c]

Notes The append/3 predicate is a classic Prolog program, and is widely used for joining and splitting lists: however, in some Prolog implementations it is not a built-in predicate, and "foreign" source files might contain a definition of append/3. In order to avoid errors, such references must be renamed or removed before loading files into **WIN-PROLOG**.

---

cat/3

join or split atoms or strings

cat(Parts, Whole, Joins)
?Parts <list> or <variable>
?Whole <string> or <atom> or <variable>

?Joins <list> or <variable>

Comments This predicate can be used to join an arbitrary number of strings or atoms together, or to split an atom or string into an arbitrary number of parts.

When joining (concatenating) text items, Parts must be bound to a list containing atoms or strings (but not both), and Whole and Joins must be unbound variables. The predicate succeeds by returning a single atom or string in Whole, together with a list of integers in Joins, which describes where to split the resulting atom or string in order to restore the original list.

When splitting (separating) an atom or a string into a list of components, Parts must be an unbound variable, and Whole and Joins must be bound to an atom or string and a list of integers respectively. The predicate succeeds by returning the list of atoms or strings that is obtained by splitting the given atom or string at the specified split points.

Examples The following call concatenates a series of atoms, returning a single atom and list of split points:

?- **cat([the,quick,brown,fox], A, J).** <enter>
A = thequickbrownfox ,
J = [3,5,5]

The following call splits a string at the specified offsets:

?- **cat(L, `jumpsoveralazydog`, [5,4,1,4]).** <enter>
L = [`jumps`,`over`,`a`,`lazy`,`dog`]

Notes The output text type generated by cat/3 (ie, atom or string) is determined by the text type that was input.

---

forall/2

test that a given goal is true for all cases of another goal

forall(Goal1, Goal2)
+Goal1 <goal>
+Goal2 <goal>

Comments This succeeds if for all solutions of Goal1, Goal2 is also true.

Examples The following call uses member/2 and atom/2 to check that all members of the given list are atoms:

?- **forall(member(X,[a,b,c]), atom(X)).** <enter>
X = _

Conversely, the following call fails because one of the list elements is an integer, rather than an atom:

?- **forall(member(X,[a,b,3]), atom(X)).** <enter>
No

Notation Conventions

Predicate Definitions
When predicate definitions are given, the functor, arguments and positions of the arguments of the predicate are shown as a template such as:
`foo(+Arg1, ?Arg2, -Arg3)`

This defines a predicate called .foo. that can take three arguments. The character that precedes each argument name is a mode declaration.

Mode Declarations
The possible "mode declarations" characters, and their meanings, are given in *Table 1 - mode declaration symbols*:

+	Denotes an input argument. It must be instantiated by the timethe predicate is called.
-	Denotes an output argument. The argument must be an uninstantiated variable when the predicate is called. If the predicate succeeds, the argument will be bound to the return value.
?	Denotes an input or output argument. The argument may be instantiated or uninstantiated

Table 1 - mode declaration symbols

Prolog Listings
Listings of Prolog programs and examples of Prolog queries are shown in 'Courier New' font.

The text that you actually type in is often shown in **bold**. Text that is output by ***WIN-***PROLOG and supplementary comments are shown in plain text.

```
?- X = [this,is,a,'PROLOG',list].
```

Horizontal ellipses (.) are used as a shorthand in examples to indicate that any number of items may be entered.
`foo( arg1, arg2, …, argn ) (n < 1)`
denotes a compound term with at least one argument.

Argument References
When the arguments that appear in the predicate templates are referred to in the body text, they appear capitalized and *italicised*.

Prolog (Logical) Variable Names

A logical variable in Prolog is an alphanumeric sequence of characters beginning with an upper case letter (A-Z) or an underscore ('_'). The alphanumeric sequence can include '_' and characters with character codes above 127. For example, the following are variable names:

```
Anything _var _1 X Var1
```

Quoting with single quotes overrides the variable name convention. For example the following are both quoted atoms:

```
'Anything' '_var'
```

Quoted atoms and atoms can be the names of global variables in VisiRule, or data items.

An underscore on its own is an anonymous variable.

Integers
An integer is a number with no fractional part. It is written as a sequence of digits, optionally preceded by a minus sign (-). Note that in ***WIN***-PROLOG an integer is in the range -2147483648 to 2147483647 (7FFFFFFFh).

The plus sign (+) must not be used to denote a positive integer. All positive integers are written without a leading sign character. For example:

```
0 1 9821 -10 -64000
```

Floating Point Numbers
A floating point number is written as an *optional* minus sign (-) followed by a sequence of one or more digits followed by a decimal point (.) followed by one or more digits, *optionally* followed by an exponent. An exponent is written as e (or E) followed by an optional minus sign followed by one to three digits.

As with integers, the plus sign (+) must not be used to denote a positive floating point number. For example:

```
1.0 246.8091 -12.3 20.003e-10 -1.3E102
```

The following are *not* floating point numbers:

```
.9        % does not start with a digit
3e-22     % no decimal point
34.1 e3   % contains a space before the 'e'
-.7       % no digit after the minus sign
56.1e4.8  % exponent is not an integer
23.       % no digit after the decimal point
```

Atoms

Atoms are text names that are used to identify data, programs, modules, files, windows, and so on.

The maximum length of an atom is 1024 bytes (this does not necessarily mean 1024 characters as ***WIN-***PROLOG supports Unicode). There main two types of atoms are alphanumeric, and quoted atoms.

Alphanumeric Atoms

An alphanumeric atom is written as a lower case letter (a-z) followed by a sequence of zero or more alphabetic characters (A-Z,a-z), digits (0-9) or underscores (_).

Note that characters with character codes above 127 are treated as lower case letters in alphanumeric atoms. For example:

```
apple a1 apple_cart test_1_case
foo123 f_T1 fred longTable
```

Quoted Atoms

A quoted atom is any sequence of characters surrounded by single quotes. To insert a single quote character in a quoted atom use two adjacent single quote characters: ''

The tilde character (~) is used within quoted atoms as an escape character. Tilde followed by a printable character in the range '@' to '_' is used to represent a control character. For example:
'~I'
represents ctrl-I.

The tilde character can also be followed by a hexadecimal integer within brackets representing the character code of a character. This can be useful for inserting characters with a character code greater than 7Fh (127).

To insert a tilde in a quoted atom use ~~.
Examples

```
'Apple' '123' '~(0)' 'hello world'
'~Ibold~M~J' '~(41)' '~(FFFFffff)' 'don''t care'
```

Lists

A list (which is a powerful recursive data structure) is a sequence of terms of the form:

$[t\,1,t\,2\,,\ldots,tn]\ n \varepsilon\ 0$

The term *ti* is the *i*'th element of the list. It can be any type of Prolog term. The simplest form of list is the empty list (*n = 0*):

```
[]
```

The following example is a four element list – the first element of which itself is a list:

```
[[a,list,of,lists],and,numbers,[1,2,3]]
```

Unknown elements of a list can be represented by variables. For example:

```
[X,Y,Z]
```

We also represent a list using the notation:

```
[ t1, t2, …, ti | Variable ] i ≥ 1
```

This list pattern represents a list that begins with the terms $t_1,t_2,..,t_i$ with the remainder of the list (the tail) denoted by *Variable*.

For example the list pattern:

```
[Head|Tail]
```

could be unified using Prolog's pattern matching algorithm with the list:

```
[1,2,3,4]
```

to give the variable bindings:

```
Head = 1
Tail = [2,3,4]
```

Arithmetic

***WIN*-**PROLOG supports mixed integer and double precision floating point arithmetic.

The LPA philosophy is that since integers and floating point numbers with no significant decimal places are logically the same, there should be no distinction between these in a high-level language like Prolog: effectively there should only be one numerical data type. The only reason integers are supported by ***WIN*-**PROLOG is for efficiency.

In ***WIN*-**PROLOG the conversion between integers and floating point numbers is transparent to user programs and occurs inside the arithmetic handler used by *is/2* and other predicates. Prior to a calculation, any integers are converted into floating point numbers, and afterwards the result is converted to an integer if possible. One exception is the addition (or subtraction) of two integers. Wherever possible this is done using integer arithmetic for speed.

Integers in ***WIN*-**PROLOG are represented in 32-bit two's complement format with a range of -2147483648 to 2147483647 (7FFFFFFFh). Floating point numbers are represented using the IEEE double precision format. This gives a precision of about 15 significant digits, and a range of 2.2e-308 to 1.7e308.

Rounding errors will invariably occur during certain operations because many decimal fractions have no direct binary representation. These errors are normally confined to the 14th or 15th digit. No attempt is made to round results to fewer decimal places.

For example, if the result of a calculation is the value 1.9999999999997 this value would not be converted to the integer 2; however there are functions to perform such rounding explicitly.

Predicates Related to Arithmetic

</2	*expression less than*
=:=/2	*expression equality*
=</2	*expression less than or equal*
=\=/2	*expression inequality*
>/2	*expression greater than*
>=/2	*expression greater than or equal*
is/2	*expression evaluator*
seed/1	*seed the random number generator*

Arithmetic Expressions

Arithmetic is performed by a number of built-in predicates that take arithmetic expressions as arguments. The most common way to perform arithmetic is using the *is/2* predicate.

An arithmetic expression can be one of the following:

- A number (integer or floating point).
- A list of the form [*X*] where *X* is a number. This allows single character strings to appear in expressions (e.g. `"a"`).
- A function. A function is represented by a compound term whose functor denotes the type of function, and whose argument(s) is itself an expression. Only certain pre-defined functions are allowed in an expression these are described in Tables 2 - 7 below.
- A bracketed expression of the form (*Expr*), where *Expr* is itself an expression.
- A variable that must have been bound to one of the above by the time the expression is evaluated. (If by the time the expression is evaluated it contains an unbound variable, a "Control Error" will be generated.)

Examples

The following are all legal arithmetic expressions.

```
23
45 * 97 / 2
sin(45)
tan((3 + 4) * 5)
[90] + 3
"A"
```

Table 5 to *Table 8* outline the arithmetic functions that can be used with the *is/2* predicate.

Function	Description
X + *Y*	the sum of *X* and *Y*.
X - *Y*	the difference of *X* and *Y*.
-*X*	the negative of *X*.
X * *Y*	the product of *X* and *Y*.
X / *Y*	the quotient of *X* and *Y*.
X // *Y*	the integer quotient of *X* and *Y*. The result is truncated to the nearest integer between it and 0.
X mod *Y*	the remainder after integer division of *X* by *Y*. The result is the same sign as *X*
X ^ *Y*	*X* to the power of *Y*.
rand(X)	computes a pseudo-random floating point number between 0 and X
sqrt(*X*)	the square root of *X*.

Table 5 - basic arithmetic functions

The trigonometric functions (see *Table 6*) work in degrees. They take a single argument *X* that can itself be an expression.

Function	Description
sin(*X*)	the sine of *X* degrees
cos(*X*)	the cosine of *X* degrees.
tan(*X*)	the tangent of *X* degrees.
asin(*X*)	the arcsine of *X* in degrees.
acos(*X*)	the arccosine of *X* in degrees.
atan(*X*)	the arctangent of *X* in degrees.

Table 6 - trigonometric functions

The following functions provide ***WIN*-**PROLOG's support for logarithms.

Function	Description
aln(*X*)	e to the power of *X*.
alog(*X*)	10 to the power of *X*.
ln(*X*)	the natural logarithm of *X*.
log(*X*)	the base 10 logarithm of *X*.

Table 7- logarithmic functions

The truncation functions (see *Table 8*) can be used for such things as rounding, returning signs and determining minimum and maximum values.

Function	Description
abs(*X*)	the absolute value of *X*. e.g. abs(-3.5) returns 3.5.
fp(*X*)	the fractional part of *X*. e.g. fp(-3.5) returns -0.5.
int(*X*)	the first integer equal to or less than *X*. e.g. int(-3.5) returns -4.
ip(*X*)	the integer equal part of *X*. e.g. ip(-3.5) returns -3.
max(*X*,*Y*)	the maximum value of *X* and *Y*. e.g. max(-3.5,4). returns 4.
min(*X*,*Y*)	the minimum value of *X* and *Y*. e.g. min(-3.5,4). returns -3.5.
sign(*X*)	-1 if *X* is negative, 0 if *X* is 0, or 1 if *X* is positive. e.g. sign(-3.5) returns -1

Table 8- truncation functions