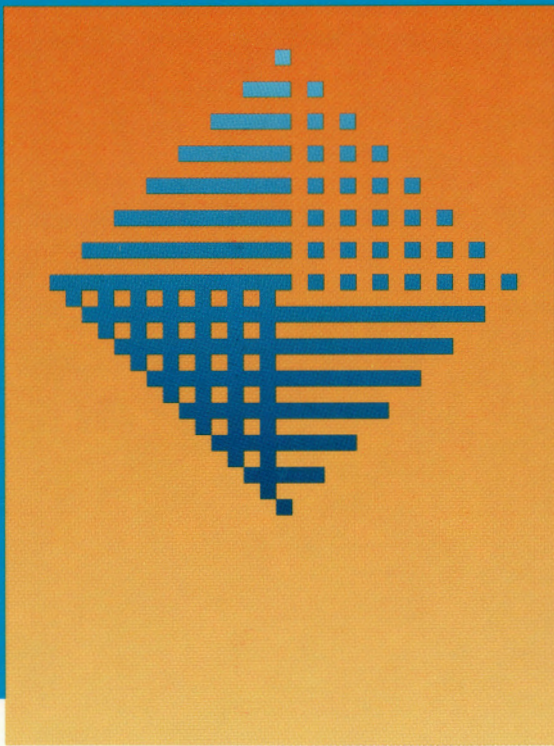
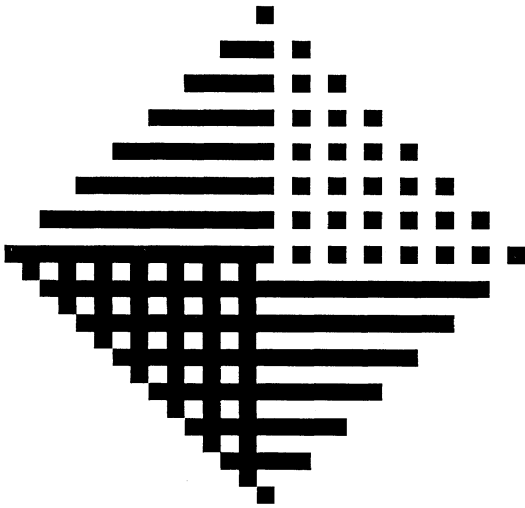


BNR Prolog

Reference Manual





BNR Prolog Reference Manual

Table of Contents

Chapter 1 Introduction	1
About This Manual	1
The Layout of the Predicate Descriptions	2
Using the Examples	3
Typographic Conventions	4
Suggested Prolog References	6
Chapter 2 Basic Language Elements	7
Syntactic Context	7
Sentences	7
Separators	7
Comments	8
Terms	9
Symbols	10
Integers	13
Floating Point Numbers	13
Intervals	14
Buckets	15
Variables	15
Lists	16
Structures	17
Constraints	17
Operators	18
Clauses	22
Chapter 3 Control	25

Chapter 4 Filters and Metapredicates	47
Basic Filters	47
Other Filters	49
Comparing Terms	53
Standard Order	53
Term Comparison Operators	54
Term Compare Predicates	55
Metapredicates	58
Chapter 5 Arithmetic	67
Functional Arithmetic	68
Arithmetic Operators	68
Arithmetic Functions	69
Arithmetic Comparisons	70
Relational Arithmetic	73
Arithmetic Operations on Intervals	75
Arithmetic Functions using Intervals	76
Arithmetic Comparison of Intervals	77
Interval Relational Expressions (is)	77
Miscellaneous Built-in Predicates	77
Chapter 6 Symbol Manipulation	81
Predicates for Manipulating Symbols	81
Chapter 7 Text Input and Output	89
Streams	89
Default Streams	90
Macintosh Pathnames	90
Input/Output Failure Conditions	90
System Errors	90
Prolog Syntax Errors	91
Stream Control Predicates	91
Character I/O Predicates	92

Term I/O Predicates	92
Chapter 8 Knowledge Base Management	119
Clauses in Contexts	120
Knowledge Base Predicates	121
Chapter 9 State Space Management	149
State Space Predicates	150
Chapter 10 Debugger	167
The Box Model	167
The Format of Port Messages	168
Interactive Port Commands	170
The Debugger	172
Debugger Predicates	174
Writing a Debugger	185
Predicates for Implementing Debuggers	187
Chapter 11 System Predicates	201
Prolog System Predicates	201
Building an Application	212
Chapter 12 Macintosh File System Access	215
Macintosh Filenames, File Types and File Creators	215
File Predicates	216
Chapter 13 Windows	231
Names and Types of Windows	231
Predicates for Handling Windows	232
Manipulating Text Window Contents	246
Predicates for Manipulating Text Windows	246
Manipulating Graph Window Contents	253
Predicates for Manipulating Graphics Windows	253

Chapter 14 Text Descriptors	257
Text Attribute Descriptors	258
Text Output Descriptors	271
Chapter 15 Graphics Descriptors	275
Graphics Attribute Descriptors	276
Graphics Output Descriptors	296
Turtle Graphics	296
Relative Output Descriptors	297
Absolute Output Descriptors	297
Miscellaneous Graphics Descriptors	297
Argument Types for the Descriptors	298
Chapter 16 Pictures	315
About Quickdraw Pictures	315
Creating and Manipulating Pictures	315
Predicates for Manipulating Pictures	316
Chapter 17 Menus	329
Predicates for Handling Menus	330
Chapter 18 User Events	345
Event Types	345
Event Reporting Priority	346
Predicates for Detecting User Events	346
Chapter 19 Predefined Dialogs	353
Predicates for Creating Dialogs	353
Chapter 20 Macintosh System Utility Predicates	367
System Utility Predicates	367

Chapter 21 External Language Interface	383
Calls to Foreign Languages	383
Accessing External Primitives	383
Data Types	385
Parameter Interface	386
Restrictions	387
Appendix A Macintosh Extended Character Set	389
Appendix B Error Messages	391
Run Errors	391
Syntax Errors	393
System Errors	395
Appendix C Compatibility with Other Prologs	397
Clause Bodies	397
Lists	397
Operators	398
Strings	398
Input and Output	398
Assert and Retract	399
All Solutions Predicates	399
Database Predicates	399
Metalogical Predicates	399
Compatibility File	399
Index	401

Chapter 1

Introduction

BNR Prolog is an extended version of Prolog for the Macintosh family of computers. Users not familiar with Prolog should read one of the textbooks listed at the end of this chapter. The user guide accompanying BNR Prolog covers the standard Prolog language. However, emphasis is placed on the extensions, such as constraints and relational arithmetic not covered in standard Prolog textbooks. The user guide also provides tutorial-style material to help users get started on the BNR Prolog system.

About This Manual

The reference manual is organized to facilitate quick access to information on language features. Each chapter covers a particular topic, with the relevant predicates presented in alphabetical order within the chapter.

Chapter 2 covers the syntax of BNR Prolog.

Chapters 3 through 9 cover the essential language predicates.

Chapter 10 describes the debugging facilities.

Chapters 11 through 20 describe the predicates associated with file and window input and output, graphics, and Macintosh system specifics.

Chapter 21 covers the external language interface.

The Appendices include a list of error messages and a list of differences (including extensions) between BNR Prolog and other Prologs.

The Layout of the Predicate Descriptions

Predicates within a chapter are listed in alphabetical order for quick reference. The first entry is the format of the predicate followed by a short verbal description. The format is the calling template and consists of the name of the predicate followed by its arguments. The layout adopted for predicate descriptions is as follows:

Arguments: A description of the arguments (if any). Special key words are used to denote the type of the arguments. These key words generally correspond to the BNR Prolog types or to compound types constructed from those types. The instantiation state of the variable is denoted by a prefix. The convention for prefixes is as follows:

"+" means "completely instantiated" (input argument).
The term is not changed during evaluation of the goal.

"-" means "variable" (output argument).
The argument is unified with a term produced during the evaluation of a goal.

"?" means either "instantiated" or "variable".

This section is always present.

Succeeds: Definition of the success semantics.
This section is always present.

Fails: Definition of the failure semantics.
This section is always present. It includes all exceptions due to type violations.

Errors: Abort semantics.
This section is optional.

Notes: Interesting or useful bits of information including caveats.
This section is optional.

Examples: A query or a sample of Prolog code that demonstrates the use of the predicate. Some tips on using the examples are given in the section below titled "Using the Examples".
This section is optional.

See Also: Gives the names of other predicates which are related to the one being described. The user is also referred to other appropriate background information.
This section is optional.

Using the Examples

The following should be noted when trying the examples:

- Information the user types in is presented in boldface and may take the form of a query, fact or rule.
- When entering facts or rules, remember to backspace over the question prompt (?-) before typing.
- Queries, facts and rules are terminated by a period (.).
- the ":-" symbol preceding a goal indicates a command.
- Pressing the *enter* key after the period "submits" the information to the system.
- Pressing the *return* key takes you to a new line.
- System responses are presented in plain text. These generally include answers to the queries. The system responds with the first answer. Press the *return* key to obtain all the remaining answers or press semicolon (;) to obtain the next answer. Press any key to terminate the question.
- Comments are delimited by /* and */. Single line comments are occasionally used. These begin with a percent sign (%) and terminate with an end of line character (that is, a return character). For example,

```
/ * this is a comment */
```

and

```
% this is a single line comment.
```

Refer to the tutorial "Using a BNR Prolog Document" in the *BNR Prolog User Guide* if you need further instructions on interacting with the Prolog system.

Typographic Conventions

The following typographical conventions are used throughout this manual:

Typewriter

Examples are displayed in this typeface. It is used to display any text which appears on the screen or in a program listing.

BNR Prolog built-in predicate names and their corresponding arguments also appear in this special typeface.

User Input

Input that the user types in is shown in boldface, while the output generated by the system is shown in *typewriter* as demonstrated by the following example.

```
?- concat(one, two, _X).  
?- concat(one, two, onetwo).  
Yes
```

keycaps

This typeface denotes a key on the Macintosh keyboard. For example,

Press the *return* key.

Italics

Italics are occasionally used to emphasize words in the text, particularly the first time a word is defined.

Italics are also used when making explicit references to titles of books.

"Quotation marks" Quotation marks are used when making cross-references to other sections of the manual or when making cross-references to chapter titles.

Quotation marks are occasionally used to highlight words in text.

The Prolog basic type symbol can be enclosed in either double or single quotation marks. Single quotation marks rather than double quotations marks have been used throughout the text when used with symbols. For example, the following is a Prolog symbol:

`'A_Symbol'`.

Suggested Prolog References

Readers who are new to Prolog may find it useful to read one of the following text books:

Bratko, I. *Prolog Programming for Artificial Intelligence*. Wokingham, England, Reading, Mass., Menlo Park, Calif., Don Mills Ont.: Addison-Wesley, 1986.

Clocksin, W. F., and Mellish, C. S. *Programming in Prolog*. 3rd ed. Berlin, Heidelberg, New York, London: Springer-Verlag, 1984.

Covington, M. A., Nute, D., and Vellino, A. *Prolog Programming in Depth*. Glenview, Ill., London: Scott, Foresman and Company, 1988.

Pereira, F. C. N., and Shieber, S. M. *Prolog and Natural-Language Analysis*. CLSI Lecture Notes, 10. Stanford: Center for the Study of Language and Information, University of Chicago Press, 1987.

Sterling, L., and Shapiro, E. *The Art of Prolog*. Cambridge, Mass., London England: The MIT Press, 1986.

Readers who will be using the interactive interface facilities provided by BNR Prolog will find it useful to refer to:

Inside Macintosh Volumes I, II, and III. Apple Computer, Inc., Addison Wesley, 1985.

Chapter 2

Basic Language Elements

Syntactic Context

To permit the extended feature set of BNR Prolog to be expressed, additional syntactic forms must be provided. Rather than attempt to add these piecemeal to a traditional Prolog syntax, a uniform, canonical syntax was developed with two objectives in mind. First, maintain a one to one relationship between the canonical syntax and the internal representation. Second, use the Edinburgh syntax model whenever possible. This second objective was included to facilitate learning of the canonical syntax by those who had been exposed to Edinburgh style Prologs, and to permit relatively straight forward porting of Edinburgh Prolog programs.

The standard output routines always write Prolog expressions in canonical form. To ease the programming task, certain license is allowed on input, but conversion to the standard canonical representation is always done.

Sentences

Prolog input streams from files or windows are sequences of sentences. Sentences consist of a Prolog term followed by a period (.), and either a space or newline.

Separators

Spaces are only significant in the following cases:

- A space is used to delimit symbols.
- A space between a name and a left parenthesis "(" is significant.

Comments

Comments may consist of any sequence of characters delimited by `/*` and `*/`. For example

```
/* This is a comment */
```

Comments may be nested

```
/* This is a /* nested */ comment */
```

or they may be on multiple lines

```
/* This is a  
multiline comment */
```

Within a single line, any text between a percent sign (`%`) and the end of line is a comment:

```
% This is a one line comment.
```

This type of comment may not be nested.

Terms

Terms are either basic or compound. Basic terms consist of constants and logic variables. Compound terms are collections of terms and may be classified as lists or structures. Figure 2-1 shows the types of terms in BNR Prolog.

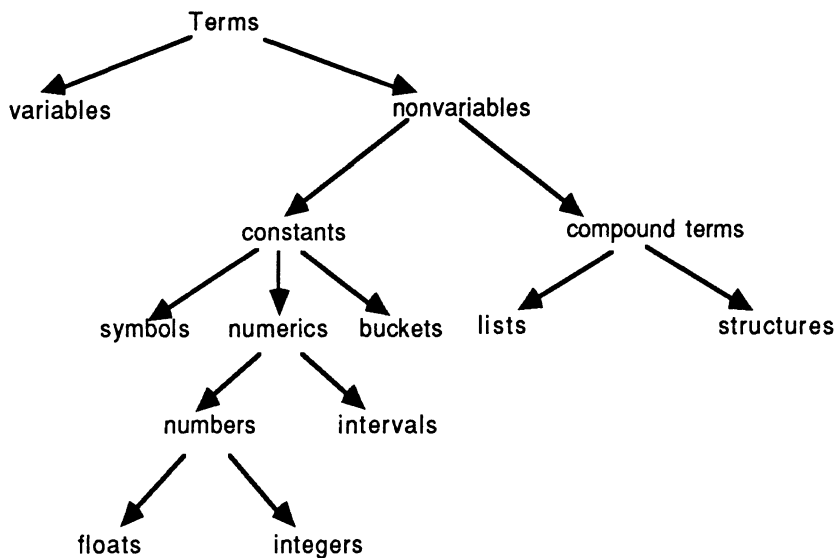


Figure 2-1. Types of BNR Prolog terms

Symbols

Symbols in BNR Prolog may be composed of any one of the following sequences:

- A sequence of alphanumerics which do not begin either with an underscore (`_`) or an uppercase letter, and are not valid numerics. The underscore and the dollar sign (`$`) are included as alphanumerics. Some examples of symbols composed of alphanumerics follow:

```
9X14
fred_100
```

- A sequence of special characters from the following set:

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

The comment sequence `/*` and `*/` is invalid in this form. Examples of this type of symbol are as follows:

```
<>++
-->
&
```

(Note that `a-->b` is a sequence of three symbols and is equivalent to `a --> b`)

- An arbitrary sequence of alphanumerics, containing at least one character, enclosed in single (`'`) or double quotation marks (`"`). A zero length sequence is represented by either a pair of single quotation marks (`''`) or a pair of double quotation marks (`""`). Examples of this type of symbol include the following:

```
'123-45'
'Kim\'s'
"Any_name"
```

An escape notation is supported for handling special ASCII characters including single and double quotation marks and for placing hexadecimal values in strings. Table 2-1 summarizes the escape sequences.

Table 2-1. Escape sequences for special characters

Special Character	Escape Sequence
new line/line feed	<code>\n</code>
horizontal tab	<code>\t</code>
backspace	<code>\b</code>
carriage return	<code>\r</code>
form feed	<code>\f</code>
backslash	<code>\\</code>
single quote	<code>\'</code>
double quote	<code>\"</code>
hexadecimal constant	<code>\hh</code>

The escape sequence `\hh`, where `h` is a character in the range 0..9 or A..F allows a hexadecimal byte value to be specified.

The Macintosh extended character set is presented in tabular form in Appendix A of this manual. The following extended ASCII characters values (shown in that table) are treated as uppercase letters:

065 - 090

128 - 134

174, 175, 184

203 - 206

217

229 - 239

241 - 244

The following ASCII character values are treated as lowercase letters:

036

097 - 122

135 - 139

185, 190, 191, 207, 216

The remaining ASCII character values are treated as special characters:

035, 038

058 - 062

064, 094, 096, 126

160 - 164

166 - 173

176 - 183

186 - 189

192 - 202

208 - 215

218 - 228

240

245 - 255

The maximum size of a symbol is 255 characters.

Symbols are case exact. For example, the symbol `fred` is not the same as `fred` and the symbol `"fred"` is not the same as `"Fred"`.

Symbols beginning with a dollar sign (\$) are local to the context in which they are defined. (Contexts are discussed the chapter titled "State Space Management" in this manual.)

Although single and double quotes are used interchangeably as symbol delimiters on input, only single quotes are used on output (as required).

Integers

An integer is a contiguous sequence of digits, optionally preceded by a minus sign (-). There should be no spaces between the sign character and the first digit. Examples of integers follow:

```
0
12344
-66311
```

Integers have a range of $-268435456 (-2^{28})$ to $268435455 (2^{28} - 1)$.

Floating Point Numbers

Floating point numbers (also called floats) are input in the following format:

```
[ - ] <digits> . <digits> [ E [ <sign> ] <digits> ]
```

Expressions appearing in square brackets are optional. The format is

- an optional minus sign character (-),
- followed by at least one digit in the range 0..9,
- followed by a decimal point,
- followed by one or more digits in the range 0..9,
- followed by an optional exponent.

The exponent, if present, is either an "E" or "e" followed by a sign character ("+" or "-"), followed by at least one digit. No spaces may occur anywhere within the floating point number.

Floating point numbers use a modified IEEE 32 bit format; the least significant 3 bits of the mantissa have been dropped providing 5 digits of precision.

Floating point numbers are output in the following exponential format:

```
[ - ] <integer part> . <fractional part> e <exponent>
```

where

- the minus sign (-) is optional,
- <integer part> is in the range 1..9,

- <fractional part> is a sequence of 4 digits,
- <exponent> is in the range -128 to +127.

If $-3 \leq \text{<exponent>} \leq 4$, then non-scientific (non-exponential) notation will be used on output. Leading and trailing zeros are omitted except for those adjacent to the decimal point.

Table 2-2 gives examples of the valid input form of some floating point numbers and their corresponding output form:

Table 2-2. Floating point input and output formats

Input Format	Output Format
9.12345	9.1234
0.00091	9.10004e-4
9.1234e-1	0.91234

The following are not valid floating point numbers:

```
.9          /* No digit before the decimal point */
9.          /* No digit after the decimal point */
99e2       /* No decimal point */
-.9        /* No digit after the sign character*/
```

Intervals

An interval is a range of values on the real number line delimited by two 32 bit floating point numbers. They provide support for relational arithmetic. (See the chapter titled "Arithmetic" in this manual for further information on interval arithmetic.)

Intervals are printed in the form `_Interval_instance` where instance is an instance number, for example, `_Interval_366236`.

Buckets

Buckets are containers for 32 bit quantities, typically pointers or handles to user defined data structures. Bucket contents only have meaning to user defined external procedures, but the containers may be unified and passed to external procedures by means of standard Prolog mechanisms. (See the chapter titled "External Language Interface" in this manual.)

Buckets are printed in the form `_Bucket_hexvalue` where `hexvalue` is the current value of the bucket contents, for example, `89AB`.

Variables

Variable names are symbols that begin either with an underscore (`_`) or an uppercase letter. Variable names may be up to 255 characters long and are case sensitive. The following are examples of variables:

```
Fred
_X100
_John_Doe
```

The underscore on its own denotes an anonymous or unnamed variable. Every occurrence of an anonymous variable in a clause represents a distinct variable.

The name `_$` is a reserved name and will be rejected as a syntax error.

Variables are always scoped within a single fact or rule. The use of `_x` in one rule is independent of the use of `_x` in any other rule.

Variable names are preserved by the system throughout computations and are used by the standard print routines. If distinct variables in any expression have the same name, for example in recursive calls, a suffix consisting of an underscore followed by digits is appended to the variable name to differentiate between them.

Lists

A *list* consists of zero or more terms (called the elements of the list) separated by commas and enclosed in square brackets. The simplest type of list is the empty list which contains zero terms. The following are examples of lists:

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

A list is often split into the initial element and the tail (the following sequence of elements). A tail variable specifies the tail of a list and is written as a variable name immediately followed by an ellipsis (..). A tail variable must always appear as the last element of a list. Indefinite lists are expressed using tail variables:

```
[_X..]  
[a, _X..]  
[_A, _B, _X..]
```

Tail variables are made uniform on input; that is, if a variable, `_X` is used as a tail variable anywhere in an expression, then all non-tail variable uses of it are transformed to `[_X..]`.

Note: The Edinburgh list, `[_H|_T]` is converted on input to `[_H, _T..]`.

Structures

Structures have the form of a symbol or a variable name, immediately followed (that is, no intervening spaces) by a parenthesized list of terms:

```
fred()  
_Fred(2, [] )  
f(2, _X..)  
_(..)
```

The symbol or variable name is called the *principal functor* of the structure. The terms in the list are the *arguments* of the structure. The number of arguments in a structure is referred to as the *arity* of the functor.

Note: A structure may have an unspecified number of arguments by using tail variables.

Constraints

Syntactically, constraints are represented as one or more terms separated by commas and enclosed in braces ({}):

```
{integer(_X), real(_Y), _X =< _Y}
```

Regarded as a data structure, a constraint is simply a structure whose functor is "{}".

Constraints are described in the chapters titled "Passive Constraints" and "Control" in the User Manual and in the chapter titled "Control" in this manual.)

Operators

Symbols defined as operators can be used to modify the parsing rules in order to improve the readability of programs. The name, type and precedence of an operator is defined by an assertion to the `op` predicate:

```
op(_Precedence, _Type, _Name).
```

Such an assertion succeeds only if all the arguments are instantiated as described below.

Any symbol can be used as an operator name. However, the tokenization rules make it easier to use symbols composed entirely of special characters. (Refer to the earlier discussion in this chapter on special characters in the extended Macintosh character set.)

These rules ensure, for example, that `p->q` is parsed as the expression

```
p -> q
```

and not as single symbol. Operator names not entirely composed of special characters must be separated from adjacent names by blanks or some other separator, such as a newline character.

The *precedence* determines the order in which operators are translated into operations. The precedence of an operator is defined by an integer in the range 0 to 1200. The lower the number the tighter the binding. If an expression contains two operators, the operation specified by the operator with the lower precedence value is executed first.

The *type* specifies the position and the associativity of the operator. The position may be either prefix, postfix or infix. A prefix operator appears before its operands, a postfix operator appears after its operands and an infix operator appears between its operands.

The *associativity* determines how operators of the same precedence are translated. The associativity of an operator can be left to right (like "+" or "-"), right to left (like "->"), or nonassociative (like "="). For example, "+" and "-" have equal precedence and associate from left to right. Thus the expression `a - b + c` is

translated as $(a - b) + c$. If two operators have the same precedence and are nonassociative, then they may not appear in the same expression. For example, the expression $a = b = c$ gives a syntax error.

The type of the operator is expressed as a symbol consisting of the characters "x", "f" and "y". "f" represents the operator and "x" and "y" represent arguments. The choice of an "x" or "y" is used to convey information about the associativity. A "y" indicates that the argument can contain operators of the same or lower precedence than the operator, and an "x" indicates that any operators in the argument must have a strictly lower precedence than the operator. Table 2-3 lists the various types and describes their significance.

Table 2-3. Type definitions for operators

Type	Position	Associativity
xfx	infix	nonassociative
xfy	infix	right to left
yfx	infix	left to right
fx	prefix	nonassociative
fy	prefix	left to right
xf	postfix	nonassociative
yf	postfix	right to left

Operators may be redefined, but

- their precedence cannot be changed
- an operator cannot be both prefix and postfix.

In expressions containing operators, the precedence rules may be overridden by using parentheses. In the expression,

```
_V is (_A + _B) * _C
```

the addition is done first, followed by the multiplication and finally the evaluation.

The increased importance of arithmetic in BNR Prolog has resulted in a minor adjustment in operator definitions. The Edinburgh arithmetic equivalence operator, "=:=", has been changed to "==", as in the C programming language. This has a small ripple effect on some of the other operators, as summarized in Table 2-4.

Table 2-4. Differences in operators in BNR Prolog and Edinburgh

Function	Edinburgh Prologs	BNR Prolog
literal identity	=	@=
literal non-identity	\=	@\=
arithmetic equality	==	=

The operators "=:=", "=\<="", and "\==" are defined to be equivalent to "==", "<>", and "@\<=" respectively. Therefore only Edinburgh "==" must be changed to "@=".

Note: One of the consequences of using the list structure for clause bodies is that the comma (,) is not an operator. Therefore, all operators, regardless of their relative precedence, bind tighter (have higher precedence) than comma.

Table 2-5 lists the predefined operators, their precedence and their type.

Table 2-5. Predefined operators

Precedence	Type	Name	Description
1200	xfx	<code>:-</code>	is true if
1100	xfy	<code>;</code>	disjunction, and else
1050	xfy	<code>-></code>	if-then
1000	xfy	<code>&</code>	explicit and
950	xfx	where	constraints
950	xfx	<code>do</code>	foreach
700	xfx	<code>=</code>	unifiability
700	xfx	<code>\=</code>	not unifiable
700	xfx	<code>is</code>	arithmetic evaluation
700	xfx	<code>==</code>	arithmetic equality
700	xfx	<code>==</code>	synonym for <code>==</code>
700	xfx	<code><</code>	arithmetic inequality
700	xfx	<code><=</code>	synonym for <code><</code>
700	xfx	<code><</code>	less than
700	xfx	<code><=</code>	less than or equal to
700	xfx	<code>></code>	greater than
700	xfx	<code>>=</code>	greater than or equal to
700	xfx	<code>@=</code>	literal identity
700	xfx	<code>@\=</code>	literal non-identity
700	xfx	<code>\=</code>	synonym for <code>@\=</code>
700	xfx	<code>@<</code>	literal less than
700	xfx	<code>@=<</code>	literal less than or identical
700	xfx	<code>@></code>	literal greater than
700	xfx	<code>@>=</code>	literal greater than or identical
700	xfy	<code>:</code>	external type specification
500	yfx	<code>+</code>	addition
500	yfx	<code>-</code>	subtraction
500	fx	<code>-</code>	unary minus
400	yfx	<code>*</code>	multiplication
400	yfx	<code>/</code>	division
400	yfx	<code>//</code>	integer division
300	yfx	<code>**</code>	exponentiation
300	xfx	<code>mod</code>	modulus

Clauses

Rules and facts have the syntactic form of a clause. Clauses have the following canonical form:

```
<structures> :- <list>.
```

The `:-` operator, read as "is true if", separates the head of the clause (the left hand side of `:-`) from its body (the right hand side of `:-`). The head of the clause is a structure. The body of the clause is a list. If a clause does not have a body, it will be coerced to a clause with a body consisting of an empty list. For example,

```
<structure> is coerced to <structure> :- [].
```

Similarly, if the body of a clause is not a list, it will be coerced to a list:

```
<structure> :- <non-list>  
is coerced to <structure> :- [<non-list>].
```

Since the body of a clause is a list, it may contain elements which are also lists. Such lists are also called blocks, since they are analogous to code blocks in procedural languages. Many of the control predicates use blocks to define the limits of their effect.

If a symbol is supplied as the head of a clause, it is coerced to a structure which consists of a functor followed by a parenthesized empty list. For example,

```
true is coerced to true().
```

Examples of clauses follow:

```
p(1) :- [].
p(2) :- [q(1)].
p(3) :- [q(1),r(2)].
p(4)      % coerced to p(4) :- [].
p(5, 6) :- q(5,6) % coerced to p(5,6) :- [q(5, 6)].
false :- fail.    %coerced to false() :- [fail].
```

A set of clauses with the same functor as the head of the clause defines a *predicate*. These clauses do not have to occur together.

If the head of a clause has a tail variable in its argument list, it is a *variadic predicate*. A variadic predicate can be written to accept any number of arguments. For example:

```
writeln(_A, B..) :- [write(_A), writeln(_B)].
writeln()      :- nl.
/* nl is a predicate which writes a new line */
```


Chapter 3

Control

The basic Prolog control sequence of goal satisfaction and failure driven backtracking is sufficient for writing any pure Prolog program. Such programs can be very inefficient or nonterminating, making it necessary to provide additional predicates which control program execution. The basic predicates, include `fail` which initiates backtracking, `cut` which limits backtracking by committing to a particular subset of possible solutions, and `failexit` which is a combination of `cut` and `fail`. The predicate `block` is used to control the scope of `cut` and is a useful concept for meta-programming applications. `freeze` and `{}` defer the execution of goals until variables are instantiated.

A second set of predicates is included to promote readability and standardize the use of certain control constructs. These include `once`, `not`, `repeat`, `;` (or), `"->"` (if-then), `"->;"` (if-then-else), and `foreach`. Finally, `findall` and `findset` are useful for generating lists of solutions; this is usually a problem when using Prolog mechanisms which do not have side-effects. `count` is used to enumerate solutions.

The following is a list of the control predicates.

!	- Edinburgh cut
;	- or
->	- if-then
->;	- if-then-else
{}	- constraints
block	- creates a block of code that can be cut
count	- enumerates solutions
cut	- controls backtracking
fail	- failure
failexit	- failure exit
findall	- constructs a list of all the solutions to a given goal
findset	- constructs a sorted list of all the solutions to a given goal
foreach	- generate each solution and test
freeze	- deferral mechanism
not	- negation by failure
once	- finds first solution
repeat	- generates infinite set of choicepoints
true	- true

Descriptions of the predicates follow.

!

*Edinburgh cut***Arguments:** None.**Succeeds:** The Edinburgh cut "!" removes all choicepoints back to and inclusive of the parent goal (with the exception of ";" and "->").**Fails:** Never fails.**Examples:**

```

/* definition for cat                                     */
cat(_X) :- !, _X = sylvester.
OK
cat(felix).
OK

?- cat(felix).
NO

/* deterministic member predicate only "finds"         */
/* or "inserts" once                                   */
member(_X, [_X, _..]) :- !.
member(_X, [_ , _Rest..]) :- member(_X, _Rest).

```

See Also: cut and fail in this chapter.

`_P ; _Q`

or

Arguments: +Goal ; +Goal

Succeeds: `_P ; _Q` (read as `_P "or" _Q`), specifies an alternation of goals. The alternation succeeds if `_P` succeeds or `_Q` succeeds on backtracking.

Fails: The alternation fails if both `_P` and `_Q` fail.

Note 1: The Prolog definition for ";" follows:

```
(_P ; _Q) :- _P.  
(_P ; _Q) :- _Q.
```

Note 2: The binding of ";" is tighter than that of the list separator ","; `[_P , _Q ; _R]` is parsed as `[_P , (_Q ; _R)]`. (See the chapter titled "Basic language Elements" in this manual for details on the precedence of operators.)

Many Prologs parse this expression as `(_P , _Q) ; _R`. To avoid confusion, it is generally a good idea to enforce precedence by using parentheses ().

Examples:

```
?- ((X = 2) ; (X = 3)).
```

```
?- ((2 = 2) ; (2 = 3)).
```

```
?- ((3 = 2) ; (3 = 3)).
```

```
YES
```

```
/* Succeeds twice
```

```
*/
```

```
?- ( true ; true ).
```

```
?- (true ; true).
```

```
?- (true ; true).
```

```
YES
```

`_P->_Q`

if-then

Arguments: +Goal -> +Goal

Succeeds: If the goal `_P` succeeds then the goal `_Q` is executed and if `_Q` succeeds then the `"->"` goal succeeds. If `_P` fails, then `_Q` is not executed and the `"->"` goal succeeds.

Fails: The `"->"` goal fails if the goal `_P` succeeds and `_Q` then fails.

Note 1: The Prolog definition for `"->"` is as follows:

```
_P -> _Q :- _P, cut('->'), _Q.  
_P -> _Q..
```

Note 2: `_P -> fail` is the same as `not(_P)`.

Note 3: `_P -> true` is the same as `once(_P ; true)`.

Examples:

```
/* Assuming p(a) succeeds but p(b) fails                               */  
/* p(a) succeeds, hello is output and goal succeeds                  */  
?- p(a) -> [nl, write(hello)].  
hello  
?- (p(a) -> [nl, write(hello)]).  
YES  
  
/* p(b) fails, hello is not output, but goal succeeds                */  
?- p(b) -> [nl, write(hello)].  
?- (p(b) -> [nl, write(hello)]).  
YES  
  
/* p(b) succeeds, but p(b) fails and goal fails                     */  
?- p(a) -> p(b).  
NO
```

See Also: `"->,"` (if-then-else) in this chapter.

`_P -> _Q ; _R`

if-then -else

Arguments: +Goal -> +Goal ; +Goal

Succeeds: The "-> ;" goal succeeds if `_P` and then `_Q` succeeds, or if `_P` fails and `_R` succeeds.

Fails: The "-> ;" goal fails if `_P` succeeds and then `_Q` fails, or if `_P` fails and `_R` fails.

Note 1: The "-> ;" predicate is defined as follows:

```
_P -> _Q ; _R :- _P, cut(';', _Q).
_P -> _Q ; _R :- cut(';', _R).
```

Note 2: The expression `_P -> _Q1 , _Q2 ; _R` is parsed as `(_P -> _Q1) , (_Q2 ; _R)`.

Examples:

```
/* Given the following facts: p(apple). p(ball). p(cat). */
:- p(apple) -> [nl,write(hello)] ; [nl,write(bye)].
hello
YES

:- p(orange) -> [nl,write(hello)] ; [nl,write(bye)].
bye
YES

/* define "translate" predicate */
translate :-
    p(apple) -> write(pomme);
    p(ball) -> write(balon);
    p(cat) -> write(chat).
```

See Also: "->" in this chapter.

```
{_Goal1, _Goal2, ...}
```

constraints

Arguments: {+Goal1,+Goal2, ...}

Succeeds: The constraint predicate succeeds if the constraints described by `_Goal1, _Goal2, ...` can be imposed on the subsequent computation. The constraints are removed on backtracking.

Fails: The constraint predicate fails if the constraints have already been violated.

Note 1: Some useful constraints follow:

1. `/* write _X whenever it is ground */
{write(_X)}.`
2. `/* prevent _X from being instantiated */
{var(_X)}.`
3. `/* constrain _X to be different from _Y */
{_X /= _Y}.`
4. `/* do q if _X becomes instantiated */
{nonvar(_X) -> q(_X)}.`
5. `/* compute _Y whenever _X is instantiated */
{_Y is _X * _X}.`
6. `/* constrain _X to be an integer < _Y */
{_X < _Y, integer(_X)}.`

Note 2: Refer to the *BNR Prolog User Guide* for a description of the use of passive constraints, active constraints and arithmetic data flow.

Examples:

```
/* constrains _X to be an integer                                */
?- [_X = 2, {integer(_X)}].
   ?- [(2 = 2), {integer(2)}].
YES

?- [{integer(_X)}, _X = 2].
   ?- [{integer(2)}, (2 = 2)].
YES

?- [_X = a, {integer(_X)}].
NO

?- [{integer(_X)}, _X = a].
NO
```

block (`_Name`, `_G1.._GN`)

creates a block of code that can be cut

Arguments: `block(+name, +term_sequence)`

Succeeds: `block` associates a name `_Name` with a list of goals and then executes the list. The name can be used as the argument to `cut` and `fail_exit` within the list.

Fails: `block` fails if `_Name` is not a symbol.

Note: `block` is useful when it is desirable (particularly, in metalevel programs, such as Prolog emulators and debuggers) to create a named executable block of code that can be cut without requiring the creation of an extra predicate. For example

```
block(fred,member(_X, _L),cut(fred), process(_X)).
```

See Also: `cut` and `fail_exit` in this chapter.

```
count(_Goal, _N)
count(_Goal, _N, _Max)
```

enumerates solutions

Arguments: count(+Goal, ?integer)
count(+Goal, ?integer, +integer)

Succeeds: The predicate `count` succeeds and unifies `_N` with the number of solutions of `_Goal`. The solutions are not necessarily distinct.

A variant of `count`, `count(_Goal, _N, _Max)` is provided to handle cases where the number of solutions is potentially very large. `_Max` must be instantiated to an integer which represents an upper limit to the number of solutions. If the number of solutions reaches `_Max`, the generation of solutions is stopped, and `_N` is instantiated to the value of `_Max`.

If `_Goal` has no solutions, `_N` is instantiated to 0.

Fails: `count(_Goal, _N)` and `count(_Goal, _N, _Max)` fail if `_N` is neither a variable nor an integer.

`count(_Goal, _N, _Max)` fails if `_Max` is an uninstantiated variable.

Note: If `_N` is initially instantiated then `count` does not stop when `_N` is exceeded.

Examples:

```
/* predicate to determine if a goal is deterministic */
deterministic(_Goal) :- count(_Goal, 1, 2).
```

cut
cut (_Name)

controls backtracking

- Arguments:** None.
cut (+symbol)
- Succeeds:** cut always succeeds and removes all choicepoints (alternatives) back to and inclusive of the current block. The current block starts with the first preceding bracket "[".
- cut (_Name) always succeeds if _Name is the name of an ancestor goal, and removes all choicepoints (alternatives) back to and inclusive of the most recent occurrence of the named goal or block.
- Fails:** cut (_Name) fails if _Name is not the name of an ancestor goal.
- Errors:** An error is generated if the argument _Name is not a symbol.
- Note 1:** The Edinburgh cut "!" cuts all choicepoints back to, and including the parent goal, and is roughly equivalent to cut (_P) where _P is the parent goal.
- Note 2:** cut is sometimes called "snips".
- Examples:**

```
/* cut removes the choicepoints of p1, but not p          */
p(_X) :- [p1, cut, P2].

/* cut removes the choicepoints of p2, but not p1 or p    */
P(_Y) :- [p1, [p2, cut], p3].
```

See Also: "!" and failexit in this chapter.

fail

failure

Arguments: None.
Succeeds: Never succeeds.
Fails: Always fails.
Note: fail can be used to force backtracking.
Examples:

```
/* fail is used to force backtracking in the          */  
/* definition of foreach                             */  
foreach(_P do _Q) :- _P, _Q, fail.  
foreach(_P do _Q).
```

See Also: true and failexit in this chapter.

failexit
failexit (_Name)

failure exit

Arguments: None.
 failexit (+symbol)

Succeeds: Never.

Fails: failexit is equivalent to the goal sequence cut, fail. This is a failure exit and all attempts to resatisfy the current block are abandoned.

failexit (_Name) is equivalent to the goal sequence cut (_Name), fail. This is a failure exit of the named predicate.

Errors: An error is generated if _Name is not a symbol.

Examples:

```

/* Both items are the same, so different must fail          */
different(_X, _X) :- failexit(different).                  */
/* Otherwise, different succeeds                            */
different(_X, _Y).                                         */

?- different(risky, risky).
NO

?-different(risque, risky)
?- different(risque, risky).
YES

/* forall solutions of the goal p do the goal q            */
forall(_P, _Q) :-                                         */
    foreach(_P do (_Q->>true; failexit(_P))).

```

See Also: cut and "!" in this chapter.

findall(*_X*, *_Goal*, *_List*)

construct a list of all the solutions to a given goal

Arguments: findall(*?Term*, *+Goal*, *?List*)

Succeeds: findall constructs a list, *_List*, consisting of all the values of *_X* such that *_Goal* is satisfied. If the attempt to satisfy *_Goal* never succeeds, then *_List* will be instantiated to the empty list.

Fails: findall fails if *_List* does not unify with the list of solutions.

Examples:

```
/* Given the facts                                     */
p(c).
OK
p(b).
OK
p(a).
OK
p(c).
OK

?- findall(_X, p(_X), _L).
   ?- findall(_X, p(_X), [c, b, a, c]).
YES

?- findall(q(_X), p(_X), _L).
   ?- findall(q(_X), p(_X), [q(c), q(b), q(a), q(c)]).
YES
```

See Also: findset in this chapter.

```
findset(_X, _Goal, _List)
```

construct a sorted list of all the solutions to a given goal

Arguments: `findset(?Term, +Goal, ?List)`

Succeeds: `findset` constructs a sorted list (duplicates are removed) `_List`, consisting of all the values of `_X` such that `_Goal` is satisfied. If the attempt to satisfy `_Goal` never succeeds, then `_List` will be instantiated to an empty list.

Fails: `findset` fails if `_List` does not unify with the list of solutions.

Examples:

```
/* Given the facts                                     */
p(c).
OK
p(b).
OK
p(a).
OK
p(c).
OK

?- findset(_X, p(_X), _1).
   ?- findset(_X, p(_X), [a, b, c]).
YES

?- findset(q(_X), p(_X), _1).
   ?- findset(q(_X), p(_X), [q(a), q(b), q(c)]).
YES
```

See Also: `findall` in this chapter.

foreach (*_P* do *_Q*)

generate each solution and test

Arguments: foreach(+Goal do +Goal)

Succeeds: For each solution of the generator *_P* do all solutions of the goal *_Q*. *foreach* always succeeds.

Fails: Never fails.

Note: The Prolog definition follows:
 foreach(*_P* do *_Q*) :- *_P*, *_Q*, fail.
 foreach(*_P* do *_Q*).

Examples:

```
/* this predicate demonstrates the use of foreach          */
/* to print the members of a list in a column            */
printcolumn(_List) :-
    foreach(member(_X, _List) do [write(_X), nl]).
```

```
freeze(_Var, _Goal)
```

deferral mechanism

Arguments: freeze(?term, ?goal)

Succeeds: If the variable `_Var` is instantiated then execute the goal; otherwise, delay the execution of the goal until the variable is instantiated. If `_Var` is uninstantiated, `freeze` always succeeds, but any operation that subsequently binds `_Var` will fail if `_Goal` fails.

Fails: `freeze` fails if the variable is instantiated but `_Goal` fails.

Note 1: The constraint goals are executed immediately after the associated variables are instantiated, but the order is unspecified.

Note 2: If the variable associated with a constraint is never instantiated, the constraint will disappear when the environment (that is, the scope) containing the variable is deallocated.

Examples:

```
/* given the following rule                                     */
test_freeze :-
    nl, write('variable: ', _V), nl,
    freeze(_V, [write('bound variable: ', _V),nl]),
    write('constrained variable: ', _V), nl,
    _V = jack; _V = jill.

:- ([test_freeze, fail] ; true).
variable: _V
constrained variable: _1
bound variable: jack
bound variable: jill
YES
```

See Also: {} in this chapter and in the *BNR Prolog User Guide*.

not (*_P*)

negation by failure.

Arguments: not(+Goal)

Succeeds: If the goal *_P* fails, then not(*_P*) succeeds.

Fails: If the goal *_P* succeeds, then not(*_P*) fails.

Note 1: The not predicate could be defined as follows:

```
not(_P) :- _P, failexit(not).
not(_P).
```

Note 2: not(not(*_P*)) is useful in cases where it is desirable to find out if *_P* is true for any possible assignment of its variables without actually binding those variables.

Note 3: If *_P* has side effects, not(not(*_P*)) has the same side effects.

Note 4: not(not(*_P*)) can be used to minimize storage.

Note 5: not(*_P*) is never a generator.

Examples:

```
/* Same as false.                                     */
?- not(true).
NO

/* Same as true.                                       */
?- not(fail).
?- not(fail).
YES

?- not(not(fail)).
NO
```

once (*_P*)

finds first solution.

Arguments: once(+Goal)

Succeeds: once finds the first solution to the goal *_P*. If backtracking occurs, no further solutions are generated.

Fails: once fails if there are no solutions to *_P*.

Note 1: The Prolog definition for once is as follows:
once(*_P*) :- *_P*, cut(once).

Note 2: When appropriate the use of once instead of cut generally results in cleaner, more storage efficient and more understandable code. It is provided as part of the basic language to encourage its use.

Examples:

```
/* using the standard member predicate                               */
?- once(member(_X, [a, b, c])).
?- member(a, [a,b,c]).
YES
```

See Also: cut and "!" in this chapter.

repeat

generates infinite set of choicepoints

Arguments: None.

Succeeds: repeat always succeeds, and when encountered during backtracking succeeds again.

Fails: repeat never fails.

Note: The prolog definition is as follows:

```
repeat.  
repeat :- repeat.
```

Examples:

```
/* Here repeat is used to read a sequence of          */  
/* numbers and generate their squares. The           */  
/* sequence is terminated by the atom end.           */  
  
squares :- repeat,  
    read(_X),  
    _X = end -> failexit(squares),  
    numeric(_X),  
    _Y is _X * _X,  
    write(_Y),  
    fail.
```

See Also: cut and failexit in this chapter.

true

true

Arguments: None.

Succeeds: Always succeeds.

Fails: Never fails.

Examples:

```
/* Equivalent to same(_X, _X):- [] */
same(_X, _X) :- true.

/* Equivalent to greek(socrates):- [] */
greek(socrates) :- true.
```

See Also: cut, fail in this chapter.

Chapter 4

Filters and Metapredicates

Basic Filters

Prolog programs are constructed from terms whose types are discerned from their syntactic form. Basic filters are predicates which can be used to test the type of a term. The classification of terms according to their corresponding filter is shown in Figure 4-1.

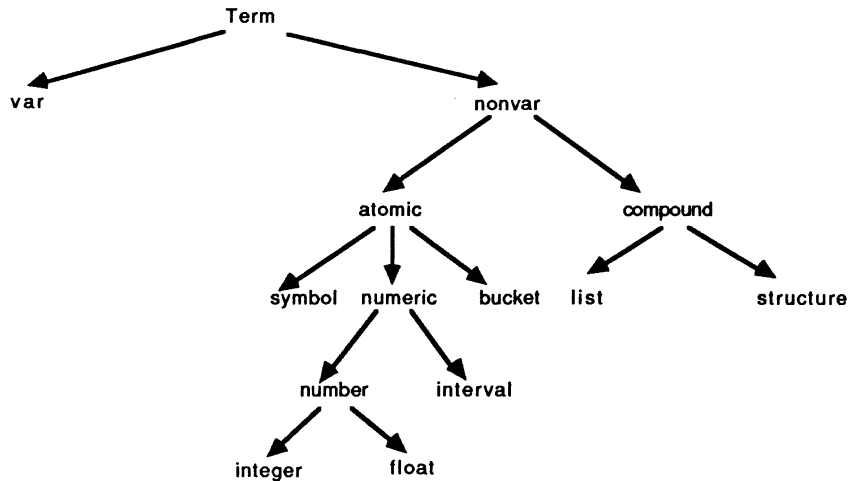


Figure 4-1. Classification of Prolog terms

Basic filters are variadic, that is, they take one or more arguments. To succeed, all the arguments must be of the type required by the filters. Table 4-1 presents a list of the basic filters. An example of a query which succeeds is presented for each filter where appropriate.

Table 4-1. Basic Filters

Name	Example
atomic	?- atomic(fred, joe, susan).
bucket	See the chapter titled "External Language Interface"
compound	?- compound([fred(2,3), [2,3,4]]).
float	?- float(4.4).
integer	?- integer(4, 5).
interval	?- [range(_X, [1, 10]), interval(_X)].
list	?- list([a,b, _X..]).
nonvar	?- [_X = fred, nonvar(_X)].
number	?- number(4.0, 4).
numeric	?- [range(_X, [1,10]), numeric(_X,4.0)].
structure	?- structure(fred(2, _X..)).
symbol	?- symbol("fred", 'fred', fred).
var	?- var(_).

Other Filters

Three other filters test some other properties of terms:

<code>acyclic</code>	- tests for cyclic structures
<code>ground</code>	- tests for variables in a term
<code>tailvar</code>	- tests for tail variables

Descriptions of these filters follow.

acyclic(_Term)

tests for cyclic structures

Arguments: acyclic(?term)

Succeeds: acyclic succeeds if the term contains no cycles (that is, the term is a finite tree).

Fails: acyclic fails if the instantiated value of the argument is a term containing cycles.

Examples:

```
?- [_X = f(_Y) , acyclic(_X)].
    ?- [(f(_Y) = f(_Y)) , acyclic(f(_Y))].
YES

?- [_X = f(2, _X) , acyclic(_X)].
NO

?- [[_X..] = [2, 3, _X..] , acyclic(_X)].
NO
```

ground(_Term)

tests for variables in a term

Arguments: ground(?term)

Succeeds: ground succeeds if _Term is ground (that is, contains no variables or tail variables).

Fails: ground fails if _Term contains one or more variables or tail variables.

Examples:

```
?- [_X = a , ground(_X)].
```

```
?- [(a = a) , ground(a)].
```

```
YES
```

```
?- [_Z = f(_V) , ground(_Z)].
```

```
NO
```

```
?- [_Z = f(_V..) , ground(_Z)].
```

```
NO
```

tailvar(*_Tailvariable..*)

tests for tail variables

Arguments: tailvar(+tailvariable)

Succeeds: tailvar succeeds if the argument is a tail variable. The term *_Tailvariable* must syntactically appear as a tail variable. (Note that the format notation does not imply that tailvar is a variadic.)

Fails: tailvar fails if the argument is not a tail variable.

Note 1: Tail variables may only occur as the last component of a list or structure.

Note 2: tailvar is the lowest level test for detecting the presence of an open-ended list structure.

Examples:

```
?- [_L = [_H, _T..], tailvar(_T..)].
   ?- [([_H, _T.. ] = [_H, _T..]), tailvar(_T..)].
YES
```

```
?- [[2,3] = [_H, _T..], tailvar(_T..)].
NO
```

Comparing Terms

The term comparison operators compare two terms in a linear order. Since terms may be of different types (for example, integers, strings, structures) it is necessary to define the order of the types, as well as an order within each type. This type ordering, called the *standard order*, is described in the following section.

Term comparison should not be confused with arithmetic comparisons. Term comparison predicates perform literal comparisons of terms in expressions; they do not evaluate those expressions.

Standard Order

The standard order for BNR Prolog is defined as follows (lowest to highest):

Variables are compared by their age, oldest first. A variable's age is independent of its name.

Tail variables are compared by their age, oldest first. A tail variable's age is also independent of its name.

Floats and *integers* are put into numeric order. Integers are put before their floating point equivalent.

Intervals are compared by age, oldest first. An interval's age is independent of its name.

Symbols are arranged in alphabetical (ASCII) order.

Lists are ordered by a term comparison of their corresponding elements, in a left-to-right order.

Structures are ordered by the name of the principal functor, and then by their arguments in a left-to-right order.

Term Comparison Operators

The term comparison operators are listed in Table 4-2.

Table 4-2. Description of comparison operators

Operator		Description
<code>_X @= _Y</code>	literal identity	Succeeds if the terms currently instantiating <code>_X</code> and <code>_Y</code> are identical.
<code>_X @\= _Y</code>	literal non-identity	Succeeds if the terms currently instantiating <code>_X</code> and <code>_Y</code> are not literally identical.
<code>_X @< _Y</code>	literal less than	Succeeds if the term <code>_X</code> is before the term <code>_Y</code> in the standard order.
<code>_X @=< _Y</code>	literal less than or identical	Succeeds if the term <code>_X</code> is not after <code>_Y</code> in the standard order.
<code>_X @> _Y</code>	literal greater than	Succeeds if the term <code>_X</code> is after the term <code>_Y</code> in the standard order.
<code>_X @>= _Y</code>	literal greater than or identical.	Succeeds if the term <code>_X</code> is not before the term <code>_Y</code> in the standard order.

Note 1: If two terms are literally identical, they must have variables at equivalent positions. For example, the query

`?- _X @= _Y.`

fails, since `_X` and `_Y` are distinct variables.

However, the query

`?- _X = _Y , _X @= Y.`

succeeds, since the variables `_X` and `_Y` are unified first.

Note 2: "@=" is equivalent to "==" in most Edinburgh Prologs.

Term Compare Predicates

<code>sort</code>	- sorts a list of terms and removes duplicates
<code>term_compare</code>	- compares terms

Descriptions of the predicates follow.

`sort(_L1, _L2)`

sorts a list of terms and removes duplicates

Arguments: `sort(+list, ?list)`

Succeeds: Sorts the elements in the list `_L1` in standard order and unifies the result with `_L2`. Multiple occurrences of the same element (as defined by the "`@=`" relationship) are removed.

Fails: `sort` fails if

- `_L1` is not a list
- `_L2` is neither a variable nor a list

Note: Sorting of `_L1` is done prior to unification with `_L2`. If `_L1` contains variables which become instantiated when `_L1` and `_L2` are unified, the `sort` relationship may no longer hold. For example,

```
?- sort([_X, _Y], [2, 1]).
```

succeeds.

Examples:

```
/* sorts in standard order                                     */
?- sort([1, 2, 2.0, 3, 1.0, 3.0], _L2).
   ?- sort([1, 2, 2.0, 3, 1.0, 3.0], [1, 1.0, 2, 2.0, 3, 3.0]).
YES

?-sort([[2,2], [1,2], [1,2,3]], _L2).
   ?- sort([[2,2], [1,2], [1,2,3]], [[1,2], [1,2,3], [2,2]]).
YES
```

term_compare(*_X*, *_Y*, *_Rel*)

compares terms

Arguments: term_compare(+term, +term, ?symbol)

Succeeds: The comparison relation between *_X* and *_Y* is *_Rel*, where possible values for *_Rel* are

'@=' if *_X* is identical to *_Y*,

'@<' if *_X* is before *_Y* in the standard order,

'@>' if *_X* is after *_Y* in the standard order.

Fails: term_compare fails if *_Rel* is neither a variable nor the comparison relation between the terms *_X* and *_Y*.

Examples:

```
?- term_compare(_X, 3, '@<').
```

```
?- term_compare(_X, 3, '@<').
```

```
YES
```

```
?- term_compare('ab', 'B', _Rel).
```

```
?- term_compare(ab, B, '@>')
```

```
YES
```

Metapredicates

Metapredicates are predicates supporting concepts which are outside the scope of first order logic. Primarily, they permit the treatment of variables as data. Further information on some uses of the metapredicates is available in the *BNR Prolog User Guide*.

The following is a list of the metapredicates.

<code>arg</code>	- index into a compound term
<code>bind_vars</code>	- binds variables to their names
<code>decompose</code>	- decomposes a cyclic structure (minimal)
<code>spanning_tree</code>	- decomposes a cyclic structure (maximal)
<code>subsumes</code>	- generality relationship
<code>termlength</code>	- relates a compound term and its length
<code>variables</code>	- collects the variables and constraints in a term

arg(*_N*, *_Term*, *_Arg*)

index into a compound term

Arguments: `arg(+integer, +term, ?term)`

Succeeds: `arg` succeeds if the *_N*th argument of *_Term* unifies with *_Arg*. (The functor of a structure is element zero (0).)

Fails: `arg` fails if

- the *_N*th argument of *_Term* is not unifiable with *_Arg*
- *_N* is not an integer
- *_Term* is not a compound term

Note: `arg` permits lists to be accessed as arrays. In some cases this may result in significant performance gains.

Examples:

```
?- arg(2, [canada, america, britain], america).
?- arg(2, [canada, america, britain], america).
YES
```

```
?- arg(2, country(canada, america, britain), _X).
?- arg(2, country(canada, america, britain), america).
YES
```

```
?- arg(0, country(canada, america, britain), _X).
?- arg(0, country(canada, america, britain), country).
YES
```

```
?- arg(3, country(canada, america, _X), britain).
?- arg(3, country(canada, america, britain), britain).
YES
```

```
?- arg(2, country, _X).
NO
```

bind_vars (*_Term*)

binds variables to their names

Arguments: bind_vars(+term)

Succeeds: Binds all variables in *_Term* to their canonical names. *_Term* is then ground.

Fails: Never fails.

Note: The canonical name of a variable is a symbol which starts with an underscore (for example, *_Name*). This name is usually given to the variable by the user. However, if distinct variables have the same name in any given expression, the system modifies the name by appending a suffix consisting of an underscore (*_*) followed by digits. Names of variables beginning with a capital letter will be prefixed with an underscore and disambiguated as necessary.

Examples:

```
/* print_column prints a list in a column                */
/* The list may contain variables                       */
/* "bind_var" stabilizes the names in the context      */
/* of _X while the not(not( undoes the binding        */
/* made by "bind_var"                                  */
prt_column(_X) :-
    [not(not([bind_var(_X), $print_col(_X)]))].

/* print subroutine                                     */
$print_col([]) :- nl.
$print_col([_X, _Xs..]) :-
    [nl, write(_X), $print_col(_Xs)].
```

See Also: ground in this chapter.

```
decompose(_Term, _Tree, _Fragments)
```

decomposes a cyclic structure (minimal)

Arguments: `decompose(+compound_term, ?tree, ?list)`

Succeeds: If `_Term` has no cycles or common subexpressions, then `_Tree` is a copy of `_Term` and `_Fragments` is the empty list. If `_Term` is cyclic then `_Tree` is a term which will become a copy of `_Term` when `_Fragments` is executed. `_Fragments` is a minimal list of unifications of the form: `[_V1 = _tree1, _V2 = _tree2, ..]`, where the `_V`'s are variables and the `_tree`'s are minimal compound terms (lists or structures). `_Tree` and the `_tree`'s in the `_Fragments` list are acyclic.

Fails: `decompose` fails if `_Term` is not a compound term.

Examples:

```
/* given the following definition for "write_cyclic"          */
write_cyclic(_x) :-
    [decompose(_x, _a, _b),
     nl, write(_a),
     nl, write(_b)
    ].

/* User imperative mode to suppress the echo                */

/* simple cyclic structure                                   */
:- [_x = f(g(_x)), write_cyclic(_x)].
_a
[(_a = f(g(_a)))].
YES
```

```
/* compound cyclic structure, decompose _x */
:- [_x = f(_y), _y = g(_x, _z), _z = h(_x, _y),
   write_cyclic(_x)].
_a
[ (_a = f(_1)), (_1 = g(_a, h(a, _1)))]
YES

/* same compound cyclic structure, decompose _y */
:- [_x = f(_y), _y = g(_x, _z), _z = h(_x, _y),
   write_cyclic(_y)].
_a
[ (_a = g[_1, h(_1, _a)]), (_1 = f(_a))]
YES

/* same compound cyclic structure, decompose fred(_z) */
:- [_x = f(_y), _y = g(_x, _z), _z = h(_x, _y),
   write_cyclic(fred(_z))].
fred[_1]
[ (_1 = g[_2, _3]), (_2 = f(_1), (_3 = h(_2, _1)))]
YES
```

```
spanning_tree(_Term, _Tree, _Fragments)
```

decomposes a cyclic structure (maximal)

Arguments: `spanning_tree(+compound_term, ?tree, ?list)`

Succeeds: If `_Term` has no cycles or common subexpressions, then `_Tree` is a copy of `_Term` and `_Fragments` is the empty list. If `_Term` is cyclic then `_Tree` is a term which will become a copy of `_Term` when `_Fragments` is executed. `_Fragments` is a maximal list of unifications of the form: `[_V1 = _tree1, _V2 = _tree2, ..]`, where the `_V`'s are variables and the `_tree`'s are maximal compound terms. `_Tree` and the `_tree`'s in the `_Fragments` list are acyclic.

Fails: `spanning_tree` fails if `_Term` is not a compound term.

Examples:

```
/* given the following definition for "write_cyclic" */
write_cyclic(_x) :-
    spanning_tree(_x, _a, _b),
    nl, write(_a), nl,
    write(_b).

/* Use imperative mode to suppress the echo */
/* simple cyclic structure */
:- [_x = f(g(_x)), write_cyclic(_x)].
f(g(_1))
[(_1 = f(g(_1)))].
YES

/* compound cyclic structure, decompose _x */
:- [_x = f(_y), _y = g(_x, _z), _z = h(_x, _y),
    write_cyclic(_x)].
f(g(_1, h(_2, _3)))
[ (_1 = f(g(_1, h(_2, _3)))), (_2 = f(g(_1, h(_2, _3)))), (_3 =
g(_1, h(_2, _3))) ]
YES
```

subsumes (*_Term1*, *_Term2*)

generality relationship

Arguments: subsumes(+term1, +term2)

Succeeds: subsumes succeeds if *_Term1* is narrower than *_Term2*, that is, any term which unifies with *_Term2* will unify with *_Term1*. (Note the reverse is not necessarily true.)

Fails: subsumes fails if *_Term1* is not as general as *_Term2*.

Note: subsumes is a filter, that is, no variables are instantiated.

Examples:

```
?- subsumes(_X, a).
```

```
?- subsumes(_X, a).
```

```
YES
```

```
?-subsumes( _X, [_Y..] ).
```

```
?- subsumes( _X, [_Y..] ).
```

```
YES
```

```
?-subsumes( [_X, a, _Y], [_U, a, []] ).
```

```
?- subsumes( [_X, a, _Y], [_U, a, []] ).
```

```
YES
```

```
?-subsumes( [_X, a, _X], [b, a, b] ).
```

```
?- subsumes( [_X, a, _X], [b, a, b] ).
```

```
YES
```

```
?-subsumes( [_X, a, _X], [b, a, c] ).
```

```
NO
```

```
?- subsumes( [_X, a, _X], [_Y, a, _Y] ).
```

```
?- subsumes( [_X, a, _X], [_Y, a, _Y] ).
```

```
YES
```

termlength(_Term, _Size, _Last)

relates a compound term and its length

Arguments: `termlength(+compound_term, ?integer, ?list)`

Succeeds: `termlength` succeeds if `_Term` is a list or structure which contains `_Size` top level elements. `_Last` is unified with an empty list if `_Term` is a definite compound term, or with the tail variable of `_Term` if it is an indefinite compound term.

Fails: `termlength` fails if

- `_Term` is not a compound term
- either of the unifications described above fail

Examples:

```
?- termlength(f(a,b,c), _S, _X).
```

```
?- termlength(f(a,b,c), 3, []).
```

YES

```
?- termlength([a,b,_C..], _S, _X).
```

```
?- termlength([a,b,_C..], 2, [_C..]).
```

YES

```
?- termlength(f(a,b,_C..), _S, []).
```

```
?- termlength(f(a,b), 2, []).
```

YES

```
?- termlength(f(a, b, _C..), _S, [e, f]).
```

```
?- termlength(f(a, b, e, f), 4, [e, f]).
```

YES

```
variables(_Term, _Vl, _Tvl, _Con)
```

collects the variables and constraints in a term

Arguments: variables(+term, ?list, ?list, ?list)

Succeeds: Unifies `_Vl` with the list of occurrences of variables in `_Term` in breadth first order. `_Tvl` is unified with the list of occurrences of tail variables in `_Term`. `_Con` is unified with the list of constraints, as a freeze expression, on variables in `_Term`.

Fails: variables fails if any of the unifications described above fails.

Examples:

```
?- variables(f(_a, _b, _a, _Xs..), _Vl, _Tvl, _C).
   ?- variables(f(_a, _b, _a, _Xs..), [_a, _b, _a],
      [[_Xs..]], []).
YES

/* use imperative mode to suppress the echo */
:- [ _x = f(_b, _x), variables(_x, _vl, _tvl, _con),
    write(_vl), nl].
[_b]
YES

/* check that there are no outstanding constraints */
?- variables(_t, _, _, []).
   ?- variables(_t, [_t], [], []).
YES

/* constrained variable example, note that variable */
/* name is modified when constraint is applied */
?- [ {integer(_X)}, variables(_X, _, _, _)].
   ?- [{integer(_1)}, variables(_1, [_1], [],
      [freeze(_1, {integer(_1)} )])].
```

Chapter 5

Arithmetic

Arithmetic is performed by predicates (for example, `is` and `==`) that take arithmetic expressions as arguments. These expressions are built from numeric constants, infix arithmetic operators, and built-in arithmetic functions. The result of evaluating an arithmetic expression may be an integer, float, or an interval. For the evaluation to succeed, each variable must be instantiated to a number or another arithmetic expression. (The acceptable ranges for integers, floats and intervals are described in the chapter titled "Basic Language Elements" in this manual.) Functions that cannot be evaluated (for example, `0 ** 0`) also cause failures.

The arithmetic infix operators, comparison operators and built-in functions which operate on floats and integers are described in the first section of this chapter titled "Functional Arithmetic". Intervals are dealt with separately in the section titled "Relational Arithmetic". Additional information on relational arithmetic can be found in the chapter titled "Relational Arithmetic" in the *BNR Prolog User Guide*.

Functional Arithmetic

Arithmetic Operators

Table 5-1. Infix arithmetic operators

Operator	Operation	Operand type	Result type
+	addition	integer,	integer if both
-	subtraction	or float	operands are
*	multiplication		integers; else float
/	division	integer or float	float
//	integer division	integer or float	integer (the result is truncated to the nearest integer)
mod	modulus	integer	integer
**	exponentiation	integer or float	integer if both operands are integers ; else float

Note 1: The operations $_X / _Y$, $_X // _Y$ and $_X \text{ mod } _Y$ fail if the value of $_Y$ is zero.

Note 2: The sign of the result of $_X \text{ mod } _Y$ is the same as the sign of $_Y$.

Note 3: The operation $_X ** _Y$ fails if both $_X$ and $_Y$ are zero.

Note 4: The operation $_X ** _Y$ may also fail if $_X$ is negative and $_Y$ is not an integer.

Arithmetic Functions

Table 5-2. Arithmetic functions

Function	Description	Result
<code>sin(_X)</code>	Sine of <code>_X</code> . (<code>_X</code> is expressed in radians.)	float
<code>cos(_X)</code>	Cosine of <code>_X</code> . (<code>_X</code> is expressed in radians.)	float
<code>tan(_X)</code>	Tangent of <code>_X</code> . (<code>_X</code> is expressed in radians.)	float
<code>asin(_X)</code>	Returns the principal value of the arcsine of <code>_X</code> in radians. ($-1 \geq _X \leq +1$)	float
<code>acos(_X)</code>	Returns the principal value of the arccosine of <code>_X</code> in radians. ($-1 \geq _X \leq +1$)	float
<code>atan(_X)</code>	Returns the principal value of the arctangent of <code>_X</code> in radians.	float
<code>abs(_X)</code>	Absolute value of <code>_X</code> .	float
<code>exp(_X)</code>	e raised to the power of <code>_X</code> .	float
<code>ln(_X)</code>	Logarithm to the base e of <code>_X</code> . (<code>_X</code> must be a positive value.)	float
<code>sqrt(_X)</code>	Returns the floating point positive square root. (<code>_X</code> must be a non-negative value.)	float
<code>integer(_X)</code>	Returns the integer part of <code>_X</code> .	integer
<code>float(_X)</code>	Returns the floating point representation of <code>_X</code> .	float
<code>floor(_X)</code>	Returns the largest integer equal to or less than <code>_X</code> .	integer
<code>ceiling(_X)</code>	Returns the smallest integer greater than or equal to <code>_X</code> . (<code>_X</code> may be an integer, or float.)	integer
<code>round(_X)</code>	Returns the closest integer to <code>_X</code> .	integer
<code>maxint</code>	Returns the largest positive integer.	integer
<code>maxreal</code>	Returns the largest floating point value.	float
<code>max(_X, _Y)</code>	Returns the maximum of <code>_X</code> and <code>_Y</code> .	integer if both <code>_X</code> and <code>_Y</code> are integers; else float.
<code>min(_X, _Y)</code>	Returns the minimum of <code>_X</code> and <code>_Y</code> .	integer if both <code>_X</code> and <code>_Y</code> are integers; else float.
<code>cputime</code>	Returns the system elapsed time.	float
<code>π/pi</code>	Returns the value of π .	float

Arithmetic Comparisons

Arithmetic comparisons are performed by the infix comparison operators. Both arguments are evaluated, using type coercions where necessary, and compared according to the operator semantics. If the relation is true, the goal succeeds. If one operand is an integer and the other a float, the integer is coerced to a float.

Table 5-3. Arithmetic comparison operators

Operator	Description
<	less than
=<	equal to or less than
==	equal to
>	greater than
>=	greater than or equal to
<>, \==	not equal

The "==" comparison operator performs a bit wise comparison on floats. Hence, the usual floating point anomalies are observed. For example, the following query fails

```
?- 1.1 * 1.1 == 1.21.  
NO
```

due to rounding errors in inexact floating point values.

`_X is _Expression`

arithmetic evaluation

Arguments: ?X is +Expression

Succeeds: If `_X` is a variable then the infix operator `is` evaluates the arithmetic expression `_Expression` and instantiates `_X` with the result. If `_X` is a numeric then the `is` operation succeeds if the value of the expression is equal to the value of `_X`.

Fails: `is` fails if

- `_Expression` is not an evaluable arithmetic expression
- `_X` is neither an evaluated variable nor a numeric
- the value of `_X` is not equal to the value of `_Expression`

Examples:

```

/* leave a space before the final period when you          */
/* have a query which ends with a numeric                  */
?- [_X is 3 * 4, 2 is _X // 5].
   ?- [(12 is (3 * 4)), (2 is (12 // 5))].
YES

/* right hand side contains a variable                      */
?- _X is _Y + 1 .
NO

/* fred is neither a variable nor a numeric                 */
?- fred is 77 .
NO

```

integer_range (*_X*, *_Lb*, *_Ub*)

integer range generator

Arguments: integer_range(?integer, +number, +number)

Succeeds: integer_range succeeds if *_X* is instantiated to an integer value between the lower bound *_Lb* and the upper bound *_Ub* inclusive. If, however, *_X* is a variable when the call is made then integer_range generates the set of integers in the range specified.

Fails: integer_range fails if

- *_X* is instantiated to a value which is not in the range specified
- either *_Lb* or *_Ub* is not an integer
- the value of the upper bound is less than the value of the lower bound

Examples:

```
/* Verify that an integer is in a given range */
?- integer_range(2, 1, 10).
?- integer_range(2, 1, 10).
YES

/* Generate integers between 1 and 3 */
?- integer_range(_X, 1, 3).
?- integer_range(1, 1, 3).
?- integer_range(2, 1, 3).
?- integer_range(3, 1, 3).
YES

/* Fails, bounds cannot be expressions */
?- integer_range(_X, [1, 2 + 2]).
NO
```

Relational Arithmetic

BNR Prolog introduces a new data type for numbers, distinct from floats or integers: the type `interval`. An interval defines a continuous range of real numbers lying between a lower and an upper bound. The bounds of an interval are floating point numbers that define its range. Any operations performed on intervals have the effect of attempting to narrow the range of the interval. An interval can only be unified with itself or an unbound variable. Intervals are created using the predicate `range` and printed using the predicate `print_interval` as described in the chapter titled "Text Input/Output". Refer to the chapter titled "Relational Arithmetic" in the *BNR Prolog User Guide* for further information on the use of intervals.

```
range(_I, [_Lb, _Ub])
```

creates or queries an interval

Arguments: range(?interval, [?number, ?number])

Succeeds: If `_I` is a variable, it is bound to an interval which lies between the upper and lower bounds specified. If the upper and lower bounds are not specified then `range` creates an interval which lies between the largest negative and the largest positive floating point values representable by the internal floating point format. If `_I` is instantiated when the call is made, `range` succeeds if `_I` lies between, or can be constrained to lie between, the bounds specified.

Fails: `range` fails if

- the interval specified does not lie between the bounds specified
- `_Lb` and `_Ub` are neither variables nor numbers
- `_I` is neither an interval nor a variable
- the value of upper bound is less than the value of the lower bound.
- `_I` is an interval with a range disjoint from `[_Lb, _Ub]`

Examples:

```
?- range(_I, [1.0, 10.0]).
   ?- range(_Interval_368264, [1.0, 10.0]).
YES

/* Fails, _I cannot be >= 4.44 and <= 2.22                                     */
?- range(_I, [4.44, 2.22]).
NO

?- range(_I, [_, _]). % Indefinite interval
   ?- range(_Interval_368376, [-3.4000e+38, 3.4000e+38]).
YES
```

Arithmetic Operations on Intervals

With the exception of integer division (`//`) and modulus (`mod`), all the arithmetic operations work with intervals.

Table 5-4. Arithmetic operations on intervals

Operator	Operation	Operand type	Result type
<code>+</code>	addition	interval and either interval, integer or float	interval
<code>-</code>	subtraction	interval and either interval, integer or float	interval
<code>*</code>	multiplication	interval and either interval, integer or float	interval
<code>/</code>	division	interval and either interval, integer or float	interval
<code>**</code>	exponentiation	interval and either integer or float	interval

The operation `_x ** _y` fails if both `_x` and `_y` are zero, or if `_y` is not an integer.

Arithmetic Functions using Intervals

Most of the arithmetic functions which operate on integers and floats also act as relations on intervals.

Table 5-5. Arithmetic relations on intervals

Function	Description	Result
<code>sin(_X)</code>	Sine of <code>_X</code> . (<code>_X</code> is expressed in radians.)	interval
<code>cos(_X)</code>	Cosine of <code>_X</code> . (<code>_X</code> is expressed in radians.)	interval
<code>tan(_X)</code>	Tangent of <code>_X</code> . (<code>_X</code> is expressed in radians.)	interval
<code>asin(_X)</code>	Returns the principal value of the arcsine of <code>_X</code> in radians. ($-1 \geq _X \leq +1$)	interval
<code>acos(_X)</code>	Returns the principal value of the arccosine of <code>_X</code> in radians. ($-1 \geq _X \leq +1$)	interval
<code>atan(_X)</code>	Returns the principal value of the arctangent of <code>_X</code> in radians.	interval
<code>abs(_X)</code>	Absolute value of <code>_X</code> .	interval
<code>sqrt(_X)</code>	Non-negative square root of <code>_X</code> .	interval
<code>exp(_X)</code>	<code>e</code> raised to the power of <code>_X</code> .	interval
<code>max(_X, _Y)</code>	Maximum of <code>_X</code> and <code>_Y</code> .	interval
<code>min(_X, _Y)</code>	Minimum of <code>_X</code> and <code>_Y</code> .	interval
<code>delta(_X)</code>	Size of <code>_X</code> .	float
<code>midpoint(_X)</code>	Arithmetic mean of range of <code>_X</code> .	float
<code>median(_X)</code>	Zero (0) if <code>_X</code> contains 0; else a value which divides the interval into subintervals containing the same number of floats. Fails if the interval contains no numbers representable as floats.	float

The inverse functions `asin`, `acos`, `atan` and `ln` can be implemented by using the function "backwards". For example,

`_X == exp(_Y)` is equivalent to `_Y == ln(_X)`.

Arithmetic Comparison of Intervals

Comparisons of expressions containing intervals are performed using the infix comparison operators listed in Table 5-3. If the comparison is successful, intervals involved in the evaluation may be narrowed. Type coercions are determined by the following rule:

If one operand is an interval and the other either an integer or a float, then the noninterval will be coerced to a "point" interval.

Interval Relational Expressions (is)

Evaluation of expressions of the form

`_v is expression`

where `_v` is a variable and `expression` contains intervals, will instantiate `_v` to an interval. Each subexpression using intervals is computed in the usual way (see the predicate description for "`_x is _Expression`" in this chapter). When a binary operation involves both an interval and either a float or an integer, the float or integer is converted to a point interval.

Miscellaneous Built-in Predicates

<code>accumulate</code>	- accumulates values between interval computations
<code>solve</code>	- forces solutions to sets of interval equations

Descriptions of the predicates follow.

accumulate(_X, _Expression)

transfers interval values between computations

Arguments: accumulate(+interval, +expression)

Succeeds: accumulate evaluates the expression (as an interval) and then adds the result to _X.

Fails: accumulate fails if

- _X is not instantiated to an interval
- _Expression is not a valid and fully instantiated arithmetic expression

Note 1: Unlike `is`, information from _X does not flow back in to the expression during this operation, and the original value of _X is not restored on backtracking. Therefore, this predicate should be used with care.

Note 2: Since the value of _X is changed by the operation, not merely narrowed, it should not be constrained by any equations.

Examples:

```
/* computes the mean over the solutions given by a generator */
mean(_X where _P, _Mean) :-
  [range(_Acc, [0, 0]), % zero accumulator
   count([_P, accumulate(_Acc, X)], _N), % compute total
   _Mean is midpoint(_Acc)/_N % compute average
  ].
```

solve (*_X*)

forces solution to sets of interval equations

Arguments: solve(+interval)

Succeeds: solve succeeds if *_X* is an interval and can be narrowed to a subinterval containing a possible solution to the current set of interval constraints on *_X*. On backtracking, successive disjoint subintervals will be generated. When solve succeeds, all intervals jointly constrained with *_X* will also be narrowed.

Fails: solve fails if

- *_X* is not an interval
- a subinterval containing a possible solution cannot be found

Note 1: The predicate solve can be used to artificially subdivide intervals in order to find solutions to sets of interval equations. See the chapter titled "Relational Arithmetic" in the *BNR Prolog User Guide* for more information on using solve.

Note 2: If *_X* is an interval and solve(*_X*) fails, then there are no solutions to the set of constraints on *_X* in the initial interval.

Examples:

```
:- range(X,_), 17 * X**256 + 35 * X**17 - 99 * X == 0,
foreach(solve(X) do [nl, print_interval(X)]).
[0.0, 0.0]
[1.005, 1.0051]
YES
```

Chapter 6

Symbol Manipulation

This chapter describes additional predicates used to process symbols. These predicates are analogous to the string procedures in C or Pascal.

Symbols are case exact, must not begin with an uppercase letter or underscore (unless they are enclosed in quotation marks) and may be any arbitrary sequence of up to 255 printable characters. Symbols may be enclosed in single or double quotation marks. The syntax of symbols is described in detail in the chapter titled "Basic Language Elements" in this manual. See the chapter titled "Filters and Metapredicates" for details of filters associated with symbols and Appendix A for a table of ASCII character codes. Conversion between symbols and other types of terms can also be performed by means of I/O operations as described in the chapter titled "Text Input and Output" in this manual.

Predicates for Manipulating Symbols

The following predicates are available for manipulating symbols.

<code>concat</code>	- concatenates two symbols
<code>lowercase</code>	- translates uppercase to lowercase
<code>name</code>	- converts between a symbol and a list
<code>namelength</code>	- returns the length of a symbol
<code>substring</code>	- extracts a substring from a symbol
<code>uppercase</code>	- replace lowercase with uppercase

Descriptions of each of the predicates follow.

concat (_Symbol1, _Symbol2, _Symbol3)

concatenates two symbols

Arguments: concat(?symbol, ?symbol, ?symbol)

Succeeds: concat succeeds if _Symbol2 is the concatenation of _Symbol1 and _Symbol2. If only _Symbol3 is instantiated, then successive values for _Symbol1 and _Symbol2 are produced on backtracking.

Fails: concat fails if

- _Symbol3, and either _Symbol1 or _Symbol2 are variable
- any of the instantiated arguments are not symbols

Examples:

```
/* Concatenate two symbols */
?- concat(one, two, _X).
   ?- concat(one, two, onetwo).
YES

/* determine the prefix */
?- concat(_X, 'Year', 'LeapYear')
   ?- concat('Leap', 'Year', 'LeapYear').
YES

/* use as a generator */
?- concat(_First, _Last, abc).
   ?- concat('', abc, abc).
   ?- concat(a, bc, abc).
   ?- concat(ab, c, abc).
   ?- concat(abc, '', abc).
YES
```

lowercase(_Symbol₁, _Symbol₂)

translates uppercase to lowercase

Arguments: lowercase(+symbol, ?symbol)

Succeeds: lowercase succeeds if _Symbol₂ is equivalent to _Symbol₁ after all uppercase letters in _Symbol₁ are replaced by their lowercase counterparts.

Fails: lowercase fails if

- _Symbol₁ is not a symbol
- _Symbol₂ is neither a symbol nor a variable

Examples:

```
/* Convert uppercase character in the symbol to          */
/* their lowercase counterparts.                          */

?- lowercase('HeLlO', _X).
   ?- lowercase('HeLlO', hello).
YES
```

See Also: uppercase in this chapter.

`name(_Symbol, _List)`

converts between a symbol and a list

Arguments: `name(?symbol, ?list)`

Succeeds: `name` succeeds if the elements of the list, `_List`, are the Apple Extended ASCII character codes (integers) composing the symbol, `_Symbol`.

Fails: `name` fails if

- `_Symbol` is neither a variable nor a symbol
- `_List` is neither a symbol of integers in the range of the Apple Extended ASCII character codes, nor a variable
- both are variables

Examples:

```
/* Convert a symbol to a list                                     */
?- name(hello, _List).
   ?- name(hello, [104, 101, 108, 108, 111]).
YES

/* Convert a list to a symbol                                    */
?- name(_Symbol, [103, 111, 97, 116]).
   ?- name(goat, [103, 111, 97, 116]).
YES

/* Check for equivalence                                        */
?- name(got, [103, 111, 97, 116]).
NO

/* Fails, some of the character codes are out                  */
/* of range                                                    */
?- name(_Symbol, [-23, 2000, 96, 97]).
NO
```

namelength(_Symbol, _Integer)

returns the length of a symbol

Arguments: `namelength(+symbol, ?integer)`

Succeeds: `namelength` succeeds if `_Integer` can be unified with the number of characters in `_Symbol`.

Fails: `namelength` fails if

- `_Symbol` is not a symbol
- `_Integer` is neither a variable nor an integer

Examples:

```
/* Find the length of the symbol                                */
?- namelength(dog, _X).
   ?- namelength(dog, 3).
YES

/* Check the length of a symbol                                */
?- namelength(dog, 3).
   ?- namelength(dog, 3).
YES
```

substring(Symbol₁, _N, _M, _Symbol₂)

extracts a substring from a symbol

Arguments: substring(+symbol, ?integer, ?integer, ?symbol)

Succeeds: substring succeeds if the substring of length `_M > 0` starting at position `_N` of `_Symbol1` is the symbol `_Symbol2`. If only `_Symbol1` is instantiated, successive values for `_N`, `_M` and `_Symbol2` are produced on backtracking.

Fails: substring fails if

- `_Symbol1` is not a symbol of length greater than zero
- `_N` and `_M` are neither variables nor integers
- `_Symbol2` is neither a variable nor a symbol of length greater than zero

Examples:

```
/* Extract the substring which starts at                               */
/* position 4 and is 3 characters in length                             */
?- substring(onetwothree, 4, 3, _X).
   ?- substring(onetwothree, 4, 3, two).
YES

/* Backtracking example                                             */
?- substring(abc, _N, _M, _X).
   ?- substring(abc, 1, 1, a).
   ?- substring(abc, 1, 2, ab).
   ?- substring(abc, 1, 3, abc).
   ?- substring(abc, 2, 1, b).
   ?- substring(abc, 2, 2, bc).
   ?- substring(abc, 3, 1, c).
YES
```

uppercase(_Symbol₁, _Symbol₂)

replaces lowercase with uppercase

Arguments: uppercase(+symbol, ?symbol)

Succeeds: uppercase succeeds if _Symbol₂ is equivalent to _Symbol₁ after all lowercase letters in _Symbol₁ are replaced by their uppercase counterparts.

Fails: uppercase fails if

- _Symbol₁ is not a symbol
- _Symbol₂ is neither a symbol nor a variable

Examples:

```
/* Replace all lowercase character with their          */
/* uppercase counterparts                             */
?-uppercase(hello, _X) .
   ?- uppercase(hello, 'HELLO').
YES
```

See Also: lowercase in this chapter.

Chapter 7

Text Input and Output

The text input/output (I/O) predicates support the reading and writing of text, that is, sequences of characters. The target (source or destination) for these predicate may be streams associated with text files or pipes, or symbols, which are limited to sequences of less than 256 characters. The use of symbols with I/O predicates permits easy and efficient conversion between internal and external (that is, text) representations of Prolog objects.

Streams

Streams are sequences of characters associated with text files, text windows or pipes, and are identified by a unique integer while the association is valid. The `open` predicate returns the identifier, which is used in subsequent `read` and `write` operations until a `close` is performed. Multiple opens of the same file are not permitted and the limit on the number of simultaneously open streams is 10.

Associated with a stream is a stream pointer. This pointer indicates the next character position from where input is taken, or to where output is placed. The first character in a stream corresponds to stream pointer position 0, the second character is position 1, and so on. There are predicates to `get` (`at`) and `set` (`seek`) the stream pointer.

A text window provides the most recent version of a file, that is, the version that is not yet committed to disk. This mapping is maintained by the Prolog system, so the user need not be aware that reading or writing is occurring to or from a window or file. The contents of a window will not be saved in the file until the user explicitly does so. Note that interactively modifying a window while it is being read as a stream, may produce strange results. Note also that the cursor position in a text window is not the same as its stream pointer.

Pipes are buffers for supporting asynchronous read and writes, that is, two internal file pointers exist (as opposed to one for normal files). Pipes act as a queue; writing occurs at the tail of the queue while reading occurs at the head. Write operations add information to the pipe, while successful read operations consume information. Both readers and writers use the same stream identifier.

Default Streams

Many I/O predicates have a form which does not specify the stream as an explicit argument. These predicates use the default input and output streams. The default input stream is a pipe which has a stream identifier of 0. The default output stream is the console window which has a stream identifier of 1. The default input stream is used to acquire interactive input from the user and the default output stream to display system output. Standard input is always submitted from the currently active window, while standard output is always written to the console window.

Macintosh Pathnames

The filename passed as an argument to the `open` predicate must be a valid Macintosh pathname. See the chapter titled "Macintosh File System Access" in this manual for the rules on naming files.

Input/Output Failure Conditions

I/O predicates fail generally for one of two types of reasons: Macintosh file system errors or Prolog I/O errors.

System Errors

A complete list of the Macintosh file system error codes is presented in Appendix A of Volume III of *Inside Macintosh*. Examples of this type of error include: I/O error, too many files open, bad filename, file is locked, and disk is full. In addition to the Macintosh file system error codes, a number of additional codes have been defined by the Prolog system. These are listed below:

MaxDocErr	=	-200; (Maximum # of documents exceeded)
UserWindErr	=	-201; (Illegal operation on a user defined window)
UnkEvErr	=	-202; (Unknown or unexpected event type seen)
WinOflwErr	=	-203; (Implementation restriction, Windows <= 32k)
UnImplErr	=	-204; (Unimplemented or inaccessible routine)
IntMMIerr	=	-205; (Internal MMI error)
ConsOpErr	=	-206; (Illegal operation on the Console window)
ProBusyErr	=	-207; (Open prolog stream can't be closed)
UserCanErr	=	-208; (User 'Cancel'.)

Prolog Syntax Errors

On input (and occasionally on output) Prolog syntax errors can be generated. These are given error code values greater than zero. Some examples of Prolog syntax errors are:

2	Incomplete term.
16	Bad character in a symbol.
32	Token is too long (input or output).

A complete list of syntax errors is given in Appendix B of this manual. Normally the only one of special interest is "incomplete term". When used with pipes it can be used to synchronize readers and writers of pipes, since a read failure on a pipe with an incomplete term does not consume any characters in the pipe. This permits a subsequent read to be satisfied after additional text has been written to the pipe.

Failure conditions specific to a predicate are noted in the description for the predicate. The general response to error conditions is predicate failure. Some predicates support an error argument; these always succeed but the error code must be checked to determine whether the I/O operation succeeded.

Stream Control Predicates

<code>at</code>	- gets the stream pointer position
<code>close</code>	- closes a stream
<code>open</code>	- opens a stream
<code>seek</code>	- sets stream position
<code>set_end_of_file</code>	- sets end of file
<code>stream</code>	- gets stream information

Character I/O Predicates

These predicates read or write text entities other than Prolog terms, namely, single characters and lines of characters.

<code>get_char</code>	- gets character from input stream
<code>nl</code>	- writes a new line to an output stream
<code>put_char</code>	- writes a character to an output stream
<code>readln</code>	- reads a string of characters from an input stream

Term I/O Predicates

The read predicates (`get_term`, `read`, `sread`) all require period (.) punctuation to delimit the end of each term read. After reading a term, the stream pointer is positioned at the first character of the next term. Reading a term can fail for a number of reasons: if there are no complete Prolog terms from the current stream pointer position to the end of the stream (for example, there may be blank lines or lines of comments trailing the last Prolog term in the stream); attempting to read more than one term from a symbol; syntax errors in complete terms (that is, those with terminating punctuation) and in incomplete terms (that is, no terminating punctuation). Reading an incomplete term from a pipe will cause the read to fail, but nothing in the pipe will be consumed. (This permits subsequent writes to the pipe to satisfy the conditions necessary for a subsequent read to be successful.) All other failure conditions consumes the data, unless there is no data to consume.

(See the chapter titled "Basic Language Elements" in this manual for the syntax of Prolog terms.)

<code>get_term</code>	- reads a term from an input source
<code>print</code>	- outputs a term to an output stream
<code>print_interval</code>	- writes an interval
<code>put_term</code>	- write a term to an output destination
<code>read</code>	- reads a sequence of terms from the default input stream
<code>sread</code>	- reads a sequence of terms from an input source
<code>write</code>	- writes a sequence of terms to an output destination
<code>writeq</code>	- formally writes a sequence of terms to an output destination
<code>write</code>	- writes a sequence of terms to the default output stream
<code>writeq</code>	- formally writes a sequence of terms to the default output stream

`at(_Stream, _Pointer)`

gets the stream pointer position

Arguments: `at(+stream , ?pointer)`

Succeeds: Unifies `_Pointer` with the stream pointer position for the stream `_Stream`. The stream pointer position is the user's current position (in characters) from the start of the stream. If the stream pointer is at the end of the stream `_Pointer` is instantiated with the symbol `end_of_file`. The pointer position for pipes is always 0.

Fails: The `at` predicate fails if

- `_Stream` is not a valid stream identifier
- `_Pointer` cannot be unified with the stream pointer position

Note: When the stream pointer is at the first character of the stream, `_Pointer` is 0 (zero); at the second character, `_Pointer` is 1 and so on.

Examples:

```
/* stream for benchmarks has just been opened */
?- [open(_X, benchmarks, read_only, 0), at(_X,
   _Pointer)].
   ?- [open(2, benchmarks, read_only), at(2, 0)].
YES

/* at can only be used for valid streams */
?- at(42, _Pointer).
NO
```

See Also: `open`, `seek` and `set_end_of_file` in this chapter.

close(*_Stream*, *_Error*)

closes a stream

Arguments: `close(+stream, ?integer)`

Succeeds: Closes the stream *_Stream*. If the close operation succeeds, *_Error* is unified with 0; if the operation is unsuccessful *_Error* is unified with a non-zero error code as described at the beginning of the chapter.

Fails: The predicate `close` fails if

- *_Stream* is not an integer
- *_Error* cannot be unified with the generated error code
- *_Stream* is the default I/O stream

Note 1: If the stream is a window, the contents will not be written to a disk file. Use one of the save commands in the **File** menu to save the window contents on disk.

Note 2: When a pipe is closed any unused data in the pipe is discarded.

Examples:

```
?- open(X, benchmarks, read_only, 0), close(X, 0).
   ?- [open(2, benchmarks, read_only, 0), close(2, 0)].
YES

/* Mac file system error -38 is 'File not open.' */
?- close(4, _Err).
   ?- close(4, -38).
YES

/* invalid stream type */
?- close(a, _Err).
NO
```

See Also: `open` and `stream` in this chapter.

```
get_char(_Char)
get_char(_Stream, _Char)
```

gets a character from an input stream

Arguments: `get_char(?char)`
`get_char(+stream, ?char)`

Succeeds: `get_char(_Char)` takes the next character from the default input stream, converts it to a symbol and unifies it with `_Char`.
`get_char(_Stream, _Char)` performs the same operation but takes the next character from the stream specified.

In either case, the stream pointer is incremented by 1.

Fails: `get_char(_Char)` and `get_char(_Char, _Stream)` fail if the converted symbol and `_Char` do not unify.

`get_char(_Char, _Stream)` fails if

- `_Stream` is not an open stream identifier
- the stream pointer position for `_Stream` is `end_of_file`.
- `_Stream` is an empty pipe and not the default input stream

Note: The default input stream is a pipe which is normally filled by characters entered from the keyboard. If there are no characters in the pipe, `get_char` waits for a character to be typed and placed in the pipe. A message `Type a Key` is displayed in the activity box of the current window.

Examples:

```
/* if q typed after query entered                               */
?- get_char(C).
   ?- get_char(q).
YES

?- get_char(0,C).
   ?- get_char(0,q).
YES
```

```
get_term(_Source, _Term, _Error)
```

reads terms from a stream or symbol

Arguments: `get_term(+source, ?term, ?integer)`

Succeeds: `get_term` reads `_Term` from the source `_Source` (stream or symbol). `_Error` is unified with an integer error code (0 = no error) and can be used to detect syntax errors, including incomplete term (`Error = 2`), as well as system I/O errors. If an incomplete term is encountered in a pipe, the stream pointer is not advanced. Permitting subsequent data entered into the pipe to complete the term. In all other cases the stream pointer is advanced, independent of whether the error code is 0.

Fails: `get_term` fails if

- `_Source` is neither a valid stream identifier nor a symbol
- `_Error` is not unifiable with the error code

Examples:

```
/* get a term from symbol, fail on any error. */
?- get_term('f(x).', _Term, 0).
   ?- get_term('f(x).', f(x), 0).
YES

/* Demonstrate use of get_term */
/* with pipes */
?- open(X, pipe, read_write_pipe, _Err).
   ?- open(2, pipe, read_write_pipe, 0).
YES

?- get_term(2, _Term, _Err).
   ?- get_term(2, [], 2).
YES

?- put_term(2, f(x), _Err).
   ?- put_term(2, f(x), 0).
YES
```

```
?- get_term(2, _Term, _Err).  
   ?- get_term(2, f(x), 0).  
YES
```

```
?- close(2, _Err).  
   ?- close(2, 0).  
YES
```

See Also: read and sread in this chapter, and Appendix B.

nl() or **nl**
nl(_Stream)

writes a new-line character to an output stream

Arguments: None
nl(+stream)

Succeeds: nl() writes a new-line character to the default output stream, while nl(_Stream) writes a new-line character to the stream specified.

Fails: nl(_Stream) fails if

- _Stream is not a valid stream identifier
- _Stream is a read_only stream

Examples:

```
?- [nl,write('hello')].  
hello  
?- [nl,write(hello)].  
YES
```

`open(_Stream, _Filename, _Mode, _Error)`

opens a stream

Arguments: `open(-Stream, +filename, +mode, ?integer)`

Succeeds: The predicate `open` instantiates `_Stream` with an identifier which is used for subsequent I/O operations on the specified stream. An integer error code is unified with `_Error`. (0 represents no error.) The mode defines the operations you can perform on the stream. The following are valid modes:

`read_only`

The stream can only be used for input; you cannot write to the stream. If a related text window already exists, its contents will be used in preference to the underlying disk file.

`read_write`

The stream is open for both input and output. If a related text window already exists, its contents will be used in preference to the underlying disk file.

`read_window`

The text window is opened for the specified stream for input only; you cannot write, using Prolog, to the window. If a window does not exist, one is created.

`read_write_window`

The text window is opened for input and output. If a window does not exist, one is created.

`read_write_pipe`

The pipe is created and opened for input and output.

In all the above cases the stream pointer is positioned at the beginning of the stream if it is opened. (If a write operation is performed, any existing information will be overwritten by new information.)

If `open` is backtracked over, the stream remains open.

- Fails:** open fails if
- the `_Filename` specified is not a valid Macintosh pathname
 - the file is currently open
 - the file cannot be opened
 - `_Error` is not unifiable with the error code

Examples:

```
/* open a file reading, fail if open fails */
?- open(_Stream, benchmarks, read_only, 0).
   ?- open(2, benchmarks, read_only, 0).
YES

/* open a pipe */
?- open(_Pipe, anyname, read_write_pipe, _Err).
   ?- open(3, anyname, read_write_pipe, 0).
```

See Also: close and stream in this chapter.

```
print (_Term)
print (_Stream, _Term)
```

outputs a term to an output stream

Arguments: `print(+term)`
 `print(+stream, +term)`

Succeeds: `print(_Term)` prints the `_Term` to the default output stream, while `print(_Stream, _Term)` prints the `_Term` to the stream specified by `_Stream`. This predicate provides a handle for user-defined pretty printing. If the user procedure `portray` is not defined then `_Term` is output, using `writeln` or `swriteln` as appropriate, and non-printable structures are converted to printable structures. If the user procedure `portray` is defined then it is used to specify the style output for `_Term`.

Fails: `print(Stream)` and `print(_Stream, _Term)` fail if the user-defined `portray` exists and fails.

`print(_Stream, _Term)` fails if

- `_Stream` is not a valid stream identifier
- `_Stream` is an input-only stream
- any component of `_Term` is longer than 255 characters

Examples:

```
?- [nl, print('a\b')].
'a\b'
?- [nl, print('a\b')].
YES

:- [nl, _X = f(_X), print(_X),].
[(_Tree where [(_Tree = f(_Tree))]).
YES
```

See Also: `swrite`, `swriteln`, `write` and `writeln` in this chapter.

```
print_interval(_Interval)
print_interval(_Stream, _Interval)
```

prints an interval

Arguments: `print_interval(+interval)`
 `print_interval(+integer, +interval)`

Succeeds: `print_interval(_Interval)` writes to the default stream, the 32 bit bounds, outward rounded interval `_Interval` in the form `[lowerbound, upperbound]`, while `print_interval(_Stream, _Interval)` writes the interval to the stream `_Stream`.

Fails: `print_interval` fails if

- `_Stream` is an invalid stream identifier
- `_Stream` in an input-only stream
- the stream pointer position is `end_of_file`
- `_Interval` is not of type `interval`

Note: The bounds output by `print_interval` may be slightly wider than those input due to internal format conversions. The interval bounds are always conservative, that is, the interval always contains the input bounds.

Examples:

```
?- [range(_I, [3.9, 4.9]), nl, print_interval(_I)].
[3.8999, 4.9001]
    ?- [range(_Interval_404612, [3.9, 4.9]), nl,
print_interval(_Interval_404612)].
YES
```

See Also: The chapter titled "Arithmetic" in this manual for further information on intervals.

`put_char(_Char)`
`put_char(_Stream, _Char)`

writes a character to an output stream

Arguments: `put_char(+char)`
`put_char(+integer, +char)`

Succeeds: `put_char(_Char)` writes the character in the single character symbol `_Char` into the default output stream.

`put_char(_Stream, _Char)` puts the character into the output stream specified by `_Stream`.

Fails: `put_char(_Char)` and `putchar(_Stream, _Char)` fail if `_Char` is not a single character symbol.

`put_char(_Stream, _Char)` fails if

- `_Stream` is not a valid stream identifier
- `_Stream` is an input-only stream

Examples:

```
?- [nl, put_char(a)].
```

```
a
```

```
?- [nl, put_char(a)].
```

```
YES
```

```
?- [nl, put_char(ab)].
```

```
NO
```

See Also: `get_char` in this chapter.

```
put_term(_Target, _Term, _Error)
```

writes a term to a target

Arguments: put_term(?integer, +term, ?integer)

Succeeds: Writes the term `_Term` followed by a term terminator (that is, a period `.`) and a space) to a target `_Target`. `_Target` is a stream or a variable which will be instantiated to a symbol containing the character sequence `.`. The output is in a form which is immediately acceptable to `get_term`, `read`, and `sread`. `_Error` is unified with an integer error code. (0 represents no error.)

Fails: put_term fails if

- `_Target` is neither a valid stream identifier nor a variable
- `_Target` is an input-only stream
- `_Error` is not unifiable with the error code
- the stream pointer position is at `end_of_file`
- any component of `_Term` is longer than 255 characters

Examples:

```
?- [nl, put_term(1, 2*3, _Err)]
(2 * 3) .
?- [nl, put_term(1, (2 * 3), 0)].
YES
?- put_term(_S, f(a, [b, c]), _Err).
?- put_term('f(a, [b, c]) .', f(a, [b, c]), 0).
YES
```

See Also: get_term and various write predicates in this chapter.

read(_Term₁, ..., _Term_n)

inputs terms from the default input stream

Arguments: read(-term₁, ..., -term_n)

Succeeds: Reads a Prolog term or a sequence of Prolog terms from the default input stream (a pipe containing text "entered" from the keyboard). Each term must be terminated with a period (.) and a space or newline.

Fails: read fails if

- the term which was input has a syntax error
- the sequence of terms specified by the argument list does not unify with sequence of terms read

Note: A period (.) followed by a return character or space character triggers error recovery if the preceding expression cannot be parsed.

Examples:

```
?- read(X, Y).  
a. b.  
?- read(a, b).  
YES
```

See Also: get_term and sread in this chapter.

readln (_Symbol)
readln (_Stream, _Symbol)

reads the sequence of characters up to the next newline from an input stream

Arguments: readln(?symbol)
readln(+stream, ?symbol)

Succeeds: readln constructs a symbol composed of all the characters from the stream pointer position of the input stream to the next end of line character. readln(_Symbol) performs the read operation on the default input stream while readln(_Stream, _Symbol) performs the read operation on the stream specified by _Stream. _Symbol is unified with the constructed string and the pointer position is moved past the end of line character. End of file is treated as an end of line but is not consumed.

Fails: readln(_Symbol) and readln(_Stream, _Symbol) fail if _Symbol is a sequence of more than 255 characters.
readln(_Stream, _Symbol) fails if

- _Stream is not a valid stream identifier
- the stream pointer for _Stream is end_of_file or the pipe is empty

Examples:

```
?- readln(Line).  
a. b.  
?- readln('a. b.').  
YES
```

seek(*_Stream*, *_Pointer*)

sets pointer address

Arguments: seek(+stream, +pointer)

Succeeds: seek moves the pointer to an offset *_Pointer* from the beginning of the stream specified by *_Stream*. *_Pointer* is either an integer (which represents a number of characters) or the symbol *end_of_file*. Advancing the stream pointer in a pipe purges any data behind the new stream pointer.

Fails: seek fails if

- *_Stream* is not a valid stream identifier
- *_Pointer* is neither an integer nor the symbol *end_of_file*
- *_Pointer* is greater than the number of characters in the stream
- *_Stream* is a default I/O stream

Note: When the stream pointer is at the first character, *_Pointer* is 0 (zero), at the second character, *_Pointer* is 1 and so on.

Examples:

```
/* open the file benchmarks */
?- open(X, benchmarks, read_only, 0).
   ?- open(2, benchmarks, read_only, 0).
YES

/* Move the pointer 4 character from the start */
?- seek(2, 4), at(2, _Pointer).
   ?- seek(2, 4), at(2, 4).
YES

/* Move pointer to the end of the file */
?-seek(2, end_of_file).
   ?- seek(2, end_of_file).
YES
```

See Also: at and set_end_of_file in this chapter.

set_end_of_file(_Stream)

sets end of file

Arguments: set_end_of_file(+stream)

Succeeds: Sets the end of stream marker at the current position in the stream _Stream. If the stream is a pipe, all data currently in the pipe will be discarded.

Fails: set_end_of_file fails if

- _Stream is not a valid stream identifier
- _Stream is the default I/O stream

See Also: at and seek in this chapter.

sread(_Source, _Term1, ..., _Termn)

reads terms from a stream or symbol

Arguments: sread(+source, ?term1, ..., ?termn)

Success: Reads a Prolog term or a sequence of terms from a stream or symbol. (Only a single term can be read from a symbol.) Each term must be terminated with a period (.) and a space or newline. Other occurrences of space or new lines are usually ignored, as are comments.

Fails: sread fails if

- `_Source` is neither a valid stream identifier nor a symbol
- a term in the sequence is not parsed successfully
- `_Source` is a stream (pipe or file) and the stream pointer reaches the `end_of_file` before the read is satisfied
- an attempt is made to read more than one term from a symbol

Examples:

```
?- sread(0,X,Y).
a. b.
?- sread(0, a, b).
YES

?- sread('1.23.', X).
?- sread('1.23.', 1.23).
YES

?- sread('a. b.', X, Y).
NO
```

See Also: read and get_term in this chapter.

stream(_Stream, _Filename, _Mode)

accesses stream information

Arguments: stream(?integer, ?filename, ?mode)

Succeeds: stream unifies _Stream, _Filename and _Mode with the stream, name and mode of the first open stream. Upon backtracking, further solutions are generated if they exist. _Filename is the full pathname of the file or pipe.

Fails: stream fails if _Stream, _Filename, and _Mode do not unify with the stream identifier, name, and mode of an open stream.

Note: Valid modes are read_only, read_write, read_window, read_write_window and read_write_pipe.

Examples:

```
?- stream(_Stream, _Filename, _Mode).
?- stream(0, default_in, read_write_pipe).
?- stream(1, 'HD:Prolog:console', read_write_window).
YES
```

swrite(_Stream, _Term₁, ..., _Term_n)

swrite(_Symbol, _Term₁, ..., _Term_n)

writes terms to a stream or symbol

Arguments: swrite(+stream, +term₁, ..., +term_n)

swrite(?symbol, +term₁, ..., +term_n)

Succeeds: swrite(_Stream, _Term₁, ..., _Term_n) writes the specified sequence of Prolog terms to the stream _Stream. The terms are written according to current operator declarations, and spaces are inserted to separate operators from their arguments. Unbound variables and tail variables are written as their names and escape sequences inside symbols are expanded. Output does not include a period or a space after each term.

swrite(_Symbol, _Term₁, ..., _Term_n) unifies _Symbol with the sequence of terms.

Fails: swrite(_Stream, _Term₁, ..., _Term_n) fails if

- _Stream is not a valid stream identifier
- _Stream is an input-only stream

swrite(_Symbol, _Term₁, ..., _Term_n)

- _Symbol is neither a symbol nor a variable
- _Symbol is a symbol which does not unify with the list of characters that would be output from swrite
- any component of any term is longer than 255 characters

Errors: See section titled "I/O Failure Conditions" in this chapter.

Examples:

```
?- [nl, swrite(1, 'answer is', 16 mod 3)].
answer is(16 mod 3)
?- [nl, write(0, 'answer is', (16 mod 3))].
YES

/* no spaces between terms                                     */
?- swrite(_S, 3, +, 4, 6).
?- swrite('3+46', 3, '+', 4, 6).
YES

?- [nl, swrite(1, 'X', X)].
X_X
?- [nl, swrite(1, 'X', _X)].
YES
```

See Also:

write, writeq, swriteq, put_term and print in this chapter, and the section on operators in the chapter "Basic Language Elements".

```
swriteq(_Stream, _Term1, ..., _Termn)  
swriteq(_Symbol, _Term1, ..., _Termn)
```

writes terms to a stream or symbol

Arguments: `swriteq(+target, +term1, ..., +termn)`
`swriteq(?symbol, +term1, ..., +termn)`

Succeeds: `swriteq(_Stream, _Term1, ..., _Termn)` writes the specified sequence of Prolog terms to the stream `_Stream`.

`swriteq(_Symbol, _Term1, ..., _Termn)` unifies `symbol` with the specified sequence of Prolog terms.

This predicate is the same as `swrite` except that it places single quotation marks around symbols when necessary. Symbols beginning with an underscore are never quoted. Escape sequences are not expanded and variables and tail variables are written as their names. This permits the written term sequence to be read with `sread`, `read`, or `get_term`, without ambiguity.

Fails: `swriteq(_Stream, _Term1, ..., _Termn)` fails if

- `_Stream` is not a valid stream identifier
- `_Stream` is an input-only stream

`swriteq(_Symbol, _Term1, ..., _Termn)` fails if

- `_Symbol` is neither a symbol nor a variable
- `_Symbol` cannot be unified with the list of characters that would be output from `swriteq`

Both variants of the predicate fail if any component of any term is longer than 255 characters.

Examples:

```
?- [nl, swriteq(1, 'answer is', 16 mod 3)].  
'answer is' (16 mod 3)  
?- [nl, swriteq(1, 'answer is', (16 mod 3))].  
YES
```

```
?- swriteq(_S,3,+,4,6).
?- swriteq('3 \'+\' 4 6 ', 3, '+', 4, 6).
YES

?- [nl,swriteq(1, 'X',X)].
'X' _X
?- [nl,swriteq(1, 'X', _X)].
YES
```

See Also:

swrite, write, writeq, put_term and print in this chapter, and bind_vars in the chapter titled "Filters and Metapredicates".

write(_Term₁, ..., _Term_n)

writes terms to the default output stream

Arguments: write(+term₁, ..., +term_n)

Succeeds: Writes the specified sequence of Prolog terms to the default output stream. The terms are written according to the current operator declaration and spaces are inserted to separate operators from their arguments. Unbound variables and tail variable are output as their canonical names. Parentheses may be output in expressions involving operators. Output does not include a period or space after each term.

Fails: write fails if any component of any term has more than 255 characters.

Note: Use writeq to put quotation marks around symbols.

Examples:

```
?- [nl, write('answer is', 16 mod 3)].
```

```
answer is(16 mod 3)
```

```
    ?- [nl, write('answer is', (16 mod 3))].
```

```
YES
```

```
/* no spaces between terms
```

```
*/
```

```
?- [nl, write(3,+,4,6)].
```

```
3+46
```

```
    ?- [nl, write(3, '+', 4, 6)].
```

```
YES
```

```
?- [nl, write('X',X)].
```

```
X_X
```

```
    ?- [nl, write('X', _X)].
```

```
YES
```

See Also: swrite, writeq, swriteq and print in this chapter, and the section on operators in the "Basic Language Elements" chapter.

```
writeq(_Term1, ..., _Termn)
```

writes a term

Arguments: writeq(+term₁, ..., +term_n)

Succeeds: writeq(_Term₁, ..., _Term_n) writes the specified sequence of Prolog terms to the default output stream.

This predicate is the same as `write` except that it places single quotation marks around symbols when necessary. Escape sequences are not expanded and variables and tail variables are written as their names.

Fails: writeq fails if any component of any term is longer than 255 characters.

Note: Standard listener output uses writeq.

Examples:

```
?- [nl, writeq('answer is', 16 mod 3)].
'answer is' (16 mod 3)
    ?- [nl, writeq('answer is', (16 mod 3))].
YES
```

```
?- [nl, writeq(3, +, 4, 6)].
3 '+' 4 6
    ?- [nl, writeq(3, '+', 4, 6)].
YES
```

```
?- [nl, writeq('X', X)].
'X' _X
    ?- [nl, writeq('X', _X)].
YES
```

See Also: `swrite`, `write`, `swriteq`, `put_term` and `print` in this chapter.

Chapter 8

Knowledge Base Management

A Prolog knowledge base is a collection of clauses stored in an area of computer memory called the *world stack* which is structured as a stack of modules called *contexts*. These contexts exist in memory as a last-in-first-out (LIFO) stack where the top of the stack is the current context (See Figure 8-1). Each context contains a set of clauses that have either been entered interactively or loaded from a file and may vary in size. The knowledge base at any given time is the union of the contexts that exist at that time.

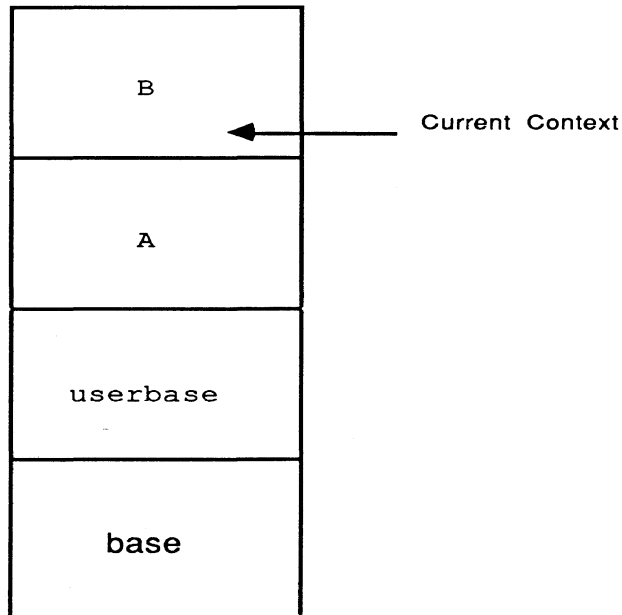


Figure 8-1. A representation of contexts

A number of predicates are provided to manage the knowledge base. Empty contexts can be created on top of the stack using `enter_context`. A set of clauses contained in a file can be loaded into a new context using `load_context`. `reload_context` is used to load a context which already exists. (If necessary, all contexts above the context being reloaded, are also reloaded. However, contexts which are not associated with files, or any dynamic assertions, are lost.)

Contexts can be removed using `exit_context`. This returns the knowledge base to the state it was in when the context was entered. The effect of any asserts, retracts or operator definitions that have taken place during the lifetime of the context are lost.

Clauses in Contexts

Names may be global, or local to a single context. Local names begin with a dollar sign (\$). Predicates with global names may have their constituent clauses defined in different contexts. In this case, the clause order is "top to bottom" within each context, with the contexts ordered from most recent to least recent (B, A, userbase, base in Figure 8-1). This strategy permits controlled refinement and overloading of predicate definitions.

Predicates which have local names are not visible outside their context. (In modular programming terms, the global predicates form the "interface procedures" of the module, while the local predicates are the implementation.) Totally independent local predicates with the same name may exist in different contexts.

Clauses may only be added to or deleted from the current context. The predicates `asserta` and `assertz` add clauses to the top and the bottom of the current context respectively. `retract` and `retractall` delete clauses from the current context. Note that since clauses can only be asserted in the current context, `assertz` behaves somewhat differently when the clause definition is spread across several contexts. The clause is actually asserted at the end of the current context which may be in the middle of the full clause chain for that predicate. A number of predicates are also available for retrieving clause definitions, clause heads and the names of various predicates.

Clause definitions can be hidden by using the predicate `hide`. This can be used as a security mechanism. Clauses can also be closed, using `close_definition`. Clauses are not subject to `assert` and `retract`. This includes implicit assertions due to context loading.

Knowledge Base Predicates

<code>assert</code>	- adds a clause to the top of the current context
<code>asserta</code>	- same as <code>assert</code>
<code>assertz</code>	- adds a clause to the bottom of the current context
<code>clause</code>	- generates clause heads
<code>clause_head</code>	- generates clause definitions
<code>close_definition</code>	- closes a predicate
<code>closed_definition</code>	- generates or verifies closed predicates
<code>consult</code>	- loads a text file into the current context
<code>context</code>	- generates or verifies context
<code>definition</code>	- generate or verifies clause definitions
<code>enter_context</code>	- creates a new context
<code>exit_context</code>	- exits a context
<code>hide</code>	- hides a predicate
<code>listing</code>	- displays predicate definitions
<code>load_context</code>	- loads a text file
<code>predicate</code>	- generates or verifies defined predicates
<code>reconsult</code>	- reloads a text file
<code>reload_context</code>	- reloads a context
<code>retract</code>	- removes clauses from the current context
<code>retractall</code>	- removes all clauses from the current context
<code>save_ws</code>	- saves the clause knowledge base
<code>symbol_name</code>	- generates or verifies the names of symbols
<code>visible</code>	- generates or verifies visible names
<code>with_context</code>	- executes predicate in context

```
assert (_Clause)
asserta (_Clause)
```

adds a clause to the top of the current context

Arguments: `assert(+clause)`
 `asserta(+clause)`

Succeeds: `asserta` adds a new clause to the top of the list of clauses in the current context. `_Clause` must be bound and unifiable with either `_Name(_Args..) :- [_Body..]` where `_Name` is a symbol, or with `_Name(_Args..) where _Name` is any symbol except ":-".

Fails: `assert` and `asserta` fail if

- `_Clause` is a variable
- `_Clause` is not unifiable with the form described above
- the functor of the clause head is the name of a closed predicate

Errors: An error is generated if the clause contains an interval, a bucket, or a looped list.

Examples:

```
/* adds a fact to the top of the current context                    */
?- assert(father(tom, susan)).
   ?- assert(father(tom, susan)).
YES

/* add a rule to the top of the current context                    */
?- assert(father(_X, _Y):-[parent(_X, _Y), male(_X)]).
   ?- assert(father(_X, _Y):-[parent(_X, _Y), male(_X)]).
YES

/* fails, the argument is variable                                    */
?- assert(_).
NO
```

See Also: `assertz` in this chapter.

assertz (*_Clause*)

adds a clause to the bottom of the current context

Arguments: assertz(+clause)

Succeeds: assertz adds a clause to the bottom of the list of clauses in the current context. *_Clause* must be instantiated and unifiable either with *_Name*(*_Args*..) :- [*_Body*..] where *_Name* is a symbol, or with *_Name*(*_Args*..) where *_Name* is any symbol except ":-".

Fails: assertz fails if

- *_Clause* is a variable
- *_Clause* is not unifiable with the form described above
- the functor of the clause is the name of a closed predicate

Errors: An error is generated if the clause contains an interval, bucket, or a looped list.

Note: Since assertz applies to the current context, the clause is only placed at the end of the clause chain if the predicate definition is confined to that context.

Examples:

```

/* add a fact to the end of the current context                               */
?- assertz( father(tom, susan)).
   ?- assertz( father(tom, susan)).
YES

/* add a fact to the top of the current context                             */
?- asserta(father(james, jenifer)).
   ?- asserta(father(james, jenifer)).
YES

/* add another fact to the bottom                                           */
?- assertz(father(adam, abel)).
   ?- assertz(father(adam, abel)).
YES

```

```
/* Query the predicate */
?- father(X, Y).
   father(james, jenifer).
   father(tom, susan).
   father(adam, abel).
YES

/* fails, the argument is not instantiated */
?- assertz(_).
NO
```

See Also: assert and asserta in this Chapter.

clause(_Clause)

generates clause definitions

Arguments: clause(?clause)

Succeeds: The clause predicate searches the knowledge base and unifies _Clause with a matching clause. Upon backtracking, further clause definitions are generated if they exist. If _Clause is instantiated it must be unifiable with _Name(_Args..) :- [_Body..] where _Name is either a variable or a symbol.

Fails: clause fails if

- _Clause is neither a variable nor a clause unifiable with the form described above
- there is no matching clause in the knowledge base
- the predicate specified by the clause is a hidden one

Examples:

```

/* Assuming standard definition for member                                */
/* (not part of preloaded system)                                       */
? - clause(member(_Args..) :- _Body) .
   ? - clause((member(_X, [_X, ..]) :- []).
   ? - clause((member(_X, [_ , _Xs..]) :- [member(_X, [_Xs..]))]).
YES

```

See Also: clause_head and definition in this chapter.

`clause_head(_Head)`

generates clause heads

Arguments: `clause_head(?clause_head)`

Succeeds: The `clause_head` predicate searches the knowledge base and unifies `_Head` with the head of a matching clause. Upon backtracking, further clause heads are generated if they exist. If `_Head` is instantiated it must be unifiable with `_Name(_Args..)` where `_Name` is either a variable or a symbol.

Fails: `clause_head` fails if

- `_Head` is neither a variable nor a clause unifiable with the form described above
- a matching clause head cannot be found

Note: `clause_head` works on hidden definitions.

Examples:

```
/* Assuming standard definition for member          */
/* (not part of preloaded system)                  */
?- clause_head(member(_Args..)).
   ?- clause_head(member(_X, [_X, _..])).
   ?- clause_head(member(_X, [_, _Xs..])).
YES
```

See Also: `clause` in this chapter.

`close_definition(_Name)`

closes a predicate

Arguments: `close_definition(+symbol)`

Succeeds: `close_definition` closes the predicate `_Name`. A closed predicate is not subject to assert or retract. Clauses may not be added to, or deleted from closed predicates.

Fails: `close_definition` fails if

- `_Name` is not a symbol
- the predicate `_Name` is not defined in the clause space

Examples:

```
/* add some facts using listener assert syntax */
dog(poodle).
OK
dog(terrier).
OK

/* close the predicate dog */
?- close_definition(dog).
   ?- close_definition(dog).
YES

/* try to assert another clause */
?- assert(dog(spaniel)). Function closed, unable to add new
clause
NO
```

See Also: `closed_definition` in this chapter.

`closed_definition(_Name)`

generates or verifies closed predicates

Arguments: `closed_definition(?symbol)`

Succeeds: If `_Name` is a variable, then `closed_definition` searches the knowledge base for the first closed predicate and generates the name of the predicate. Upon backtracking, further names will be generated if they exist. If `_Name` is a symbol, then `closed_definition` verifies the existence of the closed predicate `_Name` in the knowledge base.

Fails: `closed_definition` fails if

- `_Name` is neither a variable nor a symbol
- there are no closed predicates in the knowledge base
- `_Name` is not a closed predicate

Examples:

```
/* add some facts using listener assert syntax */
dog(poodle).
OK
dog(terrier).
OK

/* close the predicate dog */
?- close_definition(dog).
YES

/* query for closed predicates */
?- closed_definition(_X).
   ?- closed_definition(dog).
YES
```

See Also: `close_definition` in this chapter.

`consult(_Filename)`

loads a text file into the current context

Arguments: `consult(+filename)`

Succeeds: `consult` loads a text file into the current context. All clauses in the file are added to the current context. (This corresponds to Edinburgh `consult` in a single context world.)

Fails: `consult` fails if `_Filename` is not an existing text file.

Note: The preferred mechanism is `load_context`; `consult` is provided to support Edinburgh semantics.

Examples:

```
/* consult a file                                     */
?- consult('MyFile').
   ?- consult('MyFile').
YES
```

See Also: `load_context` in this chapter.

context (*_Name*)

generates or verifies contexts

Arguments context (?symbol)

Succeeds: If *_Name* is a variable, it becomes instantiated to the name of the current context. Upon backtracking, further contexts are generated if they exist. If, *_Name* is a symbol, context verifies the existence of a context of the specified name.

Fails: context fails if

- *_Name* is neither a symbol nor a variable
- *_Name* is not an existing context

Note: An ordered list of the existing contexts is also displayed in the **Contexts** menus.

Examples:

```
/* The context base is loaded by default when the Prolog      */
/* application is opened.  Userbase is the default context.   */

?- context (_X) .
   ?- context (userbase) .
   ?- context (base) .
YES

/* create a new context                                       */
?- enter_context (temp) .
   ?- enter_context (temp) .
YES

/* the new context appears at the top of the list           */
?- context (_X) .
   ?- context (temp) .
   ?- context (userbase) .
   ?- context (base) .
YES
```

```
definition(_Clause, _Context)
```

generates or verifies clause definitions

Arguments: definition(?clause, ?symbol)

Succeeds: The definition predicate searches the knowledge base and unifies `_Clause` with a matching clause and `_Context` with the name of the context in which the clause was found. Upon backtracking, further clause definitions are generated if they exist. If `_Clause` is instantiated it must be unifiable with `_Name(_Args..) :- [_Body..]` where `_Name` is either an uninstantiated variable or a symbol.

Fails: definition fails if

- `_Clause` is neither a variable nor a clause unifiable with the form described above
- `_Context` is neither a symbol nor a variable
- there is no matching clause
- the predicate specified by the clause head is a hidden one

Examples:

```
/* Assuming standard definition for member                                */
/* (not part of preloaded system)                                       */
? - definition(member(_Args..) :- _Body, _Ctxt) .
    ? - definition((member(_X, [_X, _..]) :- []), memctxt) .
    ? - definition((member(_X, [_, _Xs..]) :- [member(_X,
[_Xs..]))), memctxt) .
YES
```

See Also: clause and clause_head in this chapter.

enter_context (*_Name*)

creates a new context

Arguments: enter_context (+symbol)

Succeeds: Creates a new context *_Name* at the top of stack. This becomes the current context.

Fails: enter_context fails if *_Name* is not a symbol.

Examples:

```
/* The context base is loaded by default when the          */
/* Prolog application is opened. Userbase is the           */
/* default context created for users.                      */
*/

?- context (X)
   context (userbase).
   context (base).
YES

/* create a new context                                    */
?- enter_context (temp).
   enter_context (temp).
YES

/* the new context appears at the top of the list         */
?- context (X).
   context (temp).
   context (userbase).
   context (base).
YES
```

See Also: exit_context and context in this chapter.

exit_context (*_Name*)

exits a context

Arguments: `exit_context(+symbol)`

Succeeds: Removes the context specified by *_Name* from the stack. The context stack is returned to the state it was in prior to the corresponding `enter_context`. All intervening asserts, retracts and operator definitions that took place during the life of that context are removed.

Exiting the predefined base context succeeds, but has no effect. Exiting `userbase` has the standard semantics but the system automatically creates a new (empty) `userbase`.

Fails: `exit_context` fails if *_Name* is not an existing context.

Note: `exit_context` must be used with care within programs as it can leave dangling references if any variables are instantiated to structures or names defined in the context(s) removed, or if execution of removed code is in progress.

See Also: `context` and `enter_context` in this chapter.

hide (Name)

hides a predicate

Arguments: hide(+symbol)

Succeeds: Hides the specified predicate. Hidden predicates may be called, but their bodies cannot be seen.

Fails: Hide fails if

- Name is not a symbol
- Name is not a defined predicate

Note 1: Clause and definition predicates fail on a hidden predicates.

Note 2: Tracing, emulation and explanation facilities do not work on hidden predicates.

Note 3: The hide mechanism may be used as a security feature, since it cannot be broken using normal debugging facilities.

Examples:

```
/* Assert a clause */
animal(X) :- dog(X).
OK
?- hide(animal).
   ?- hide(animal).
YES
?- clause(animal(..):-..);
definition(animal(..):-.., ..).
NO
?- clause_head(animal(..)).
   ?- clause_head(animal(X)).
YES
:- listing(animal).
animal(X) :- [ ... ]. % Context: "userbase", Hidden Definition.
YES
```

See Also: clause_head in this chapter.

```

listing()
listing(_Name)
listing(_Name, _Context)

```

displays predicate definitions

Arguments: None.

```

listing(?symbol)
listing(?symbol, ?symbol)

```

Succeeds: `listing()` outputs all clause definitions in the current context to the console window.

`listing(_Name)` displays all clause definitions in the knowledge base for the predicate name `_Name`.

`listing(_Name, _Context)` displays all clause definitions in the knowledge base for the predicate name `_Name` in the context `_Context`.

In each case the source context for each clause is output as a comment.

Fails: `listing ()` never fails.

`listing(_Name)` fails if `_Name` is not a symbol.

`listing(_Name, _Context)` fails if `_Name` and `_Context` are not symbols.

Notes: The clauses are listed in execution search order. Individual predicates may also be listed using the **Contexts** menu.

Examples:

```

/* enter a new context and add some clauses                               */
?- enter_context(family).
   ?- enter_context(family).
YES

```

```
mother(jessica, justine).
OK

mother(geraldine, frances).
OK

/* enter another context and add another clause */
?- enter_context(tree).
   ?- enter_context(tree).
YES

mother(pat, frances).
OK

/* list the clauses for the predicate */
?- listing(mother).
mother(pat, frances) .      % Context: "tree"

mother(jessica, justine) .  % Context: family

mother(geraldine, frances) . % Context: "family"

   ?- listing(mother).
YES

?- listing.
mother(pat, frances) .      % Context: "tree"

   ?- listing().
YES

?- listing(mother, family).
mother(jessica, justine) .  % Context: family

mother(geraldine, frances) . % Context: "family"

   ?- listing(mother, family).
YES
```

`load_context(_Filename)`

loads a text file

Arguments: `load_context(+filename)`

Succeeds: `load_context` creates a new context with the same name as the specified text file, then enters the clauses from the text file into the newly created context. If the file has already been loaded then `load_context` succeeds, but has no other effect.

Fails: `load_context` fails if `_Filename` is not an existing text file.

Note: A text file may also be loaded using the **Load File..** command in the **Contexts** menu.

Examples:

```
?- load_context('MyFile').  
?- load_context('MyFile').  
YES
```

See Also: `reload_context` in this chapter.

predicate (*_Name*)

generates or verifies defined predicates

Arguments: predicate(?symbol)

Succeeds: If *_Name* is a variable, `predicate` searches the knowledge base for the most recently defined predicate and returns its name. Upon backtracking, further names are returned if they exist. If *_Name* is a symbol, then `predicate` verifies the existence of a defined predicate with the name specified.

Fails: `predicate` fails if

- *_Name* is not a symbol
- the predicate *_Name* is not defined in the knowledge base

Examples:

```
?- predicate(predicate) .  
?- predicate(predicate) .  
YES
```

```
?- predicate(_P) .  
?- predicate(accumulate) .  
?- predicate(solve) .  
?- predicate(sub_solve) .  
.  
.  
.  
YES
```

reconsult (_Filename)

reloads a text file

Arguments: reconsult(+filename)

Succeeds: reconsult loads the text file _Filename into the current context. Any predicates defined in _Filename will have their clauses in the current context retracted before the file is loaded. (This corresponds to Edinburgh consult in a single context world.)

Fails: reconsult fails if _Filename is not an existing file.

Examples:

```
?- reconsult('Mydisk:MyFile').  
?- reconsult('Mydisk:MyFile').  
YES
```

See Also: consult and reload_context in this chapter.

`reload_context(_Filename)`

reloads a context

Arguments: `reload_context(+filename)`

Succeeds: If the context is not already loaded, then `reload_context` creates a new context with the same name as the specified text file. It then enters the clauses from this file into the newly created context. If the file has already been loaded, then the contents of the context are completely replaced with the contents of `_Filename`. If necessary, all contexts which are higher on the stack (loaded after `_Filename`) will also be replaced, although any dynamic assertions in any of the contexts will be lost. Any contexts removed, but not corresponding to files, are lost.

Fails: `_Filename` is not the name of an existing text file.

Note: `reload_context` must be used with care within programs as it can leave dangling references if any variables are instantiated to structures or names defined in the context(s) removed.

Examples:

```
?- reload_context('MyFile').  
?- reload_context('MyFile').  
YES
```

See Also: `reload_context` in this chapter.

```
retract (_Clause)
```

removes clauses from the current context

Arguments: `retract(+clause)`

Succeeds: `retract` searches the current context for the first matching clause and removes it. Upon backtracking, all other matching clauses are successively removed. `_Clause` must be instantiated and unifiable with either `_Name(_Args..)` :- `[_Body..]` where `_Name` is a symbol, or with `_Name(_Args..)` where `_Name` is any symbol except ":-".

Fails: `retract` fails if

- `_Clause` is a variable
- `_Clause` is not unifiable with the form described above
- there are no matching clauses

Note 1: Execution in progress of the retracted clauses is not affected.

Note 2: Space for the clauses is not recovered; removing the context will recover space for the entire context. (State space has dynamic space recovery.)

Examples:

```
/* add some facts using listener assert syntax                               */
dog(poodle) .
OK
dog(terrier) .
OK
animal(_X) :- dog(_X) .
OK
```

```
?- retract(dog(_X)).  
   ?- retract(dog(poodle)).  
   ?- retract(dog(terrier)).  
YES  
?- retract(animal(_..):- _).  
   ?- retract(animal(_X) :- dog(_X)).  
YES
```

See Also: retractall in this chapter.

retractall(_Head)

removes all clauses from the current context

Arguments: retractall(+clause_head)

Succeeds: Removes all clauses with matching clause heads from the current context in one operation. retractall succeeds even if there are no matching clauses.

Fails: retractall fails if

- _Head is an uninstantiated variable
- the functor of the clause head is an uninstantiated variable

Examples:

```
/* add some facts using listener assert syntax */
dog(poodle) .
OK
dog(terrier) .
OK

?- retractall(dog(_X)) .
   ?- retractall(dog(_X)) .
YES
```

See Also: retract in this chapter.

save_ws (_Filename)

saves the clause knowledge base

Arguments: save_ws(+filename)

Succeeds: Saves the clause knowledge base, (that is, the stack of contexts) and configuration data (stack sizes and initial goal) as a binary image in the file specified.

Fails: save_ws fails if _Filename is not a valid Macintosh file specification.

Notes: The workspace file is loaded with the application when the file is "opened" from the desktop.

Examples:

```
?- save_ws(workspace) .  
    ?- save_ws(workspace) .  
YES
```

symbol_name (_Name)

generates or verifies the names of existing symbols

Arguments: symbol_name(?symbol)

Succeeds: If `_Name` is a variable, `symbol_name` searches the knowledge base for the most recent symbol and generates its name. Upon backtracking, further names are generated if they exist. If `_Name` is a symbol, then predicate verifies the existence of the specified symbol in the knowledge base.

Fails: symbol_name fails if `_Name` is neither a symbol nor a variable.

Examples:

```
?- symbol_name(userbase) .
   ?- predicate(userbase) .
YES

?- symbol_name(_Sn) .
   ?- symbol_name($local) .
   ?- symbol_name(base_ws) .
   ?- symbol_name(accumulate) .
   .
   .
   .
YES
```

visible (_Name)

generates or verifies visible names

Arguments: visible(?symbol)

Succeeds: If _Name is a variable, then visible searches the knowledge base for the first visible predicate (that is, one which has not been hidden using the predicate hide) and generates its name. Upon backtracking, further names are generated if they exist. If _Name is a symbol then visible verifies the existence of a visible predicate with the name specified.

Fails: visible fails if

- _Name is neither a symbol nor a variable
- there is no visible predicate _Name

Examples:

```
/* add a predicate using listener assert syntax */
dog(poodle) .
OK

?- visible(_P) .
   ?- visible(dog) .
   ?- visible(accumulate) .
   ?- visible(solve) .
YES

?- hide(dog) .
   ?- hide(dog) .

?- visible(_P) .
   ?- visible(accumulate) .
   ?- visible(solve) .
   ?- visible(sub_solve) .
YES
```

with_context (_Name, _Goal)

executes predicate in context

Arguments: with_context(+symbol, +goal)

Succeeds: Executes the goal _Goal in a new temporary context _Name (isolate from clause space side effects). The temporary context is removed whether _Goal succeeds or fails unless with_context is cut within _Goal.

Fails: with_context fails if

- _Name is not a symbol
- _Goal fails

Note: with_context must be used with care within programs as it can leave dangling references if any variables are instantiated to structures or names defined in the temporary context.

Examples:

```
/* Use with_context to discard side effects of op                               */
?- with_context(temp [nl, write('Hello')]).
Hello
?- with_context(temp, [nl, write('Hello')]).
YES
```

Chapter 9

State Space Management

State spaces are internal data bases for storing Prolog structures which are independent of the clause space. They provide user controlled storage on either a global or local basis. The important properties of state spaces include:

- Separation of data (state) from programs (clauses). This permits storing data without asserting its truth.
- Independence from the context stack.
- Automatic incremental garbage collection.
- Flexible data access.
- Support for atomic transactions including order preserving replacement.

The *global state space* is decoupled from contexts, which permits moving structures between contexts. Only one global state space exists in memory at any given time, but a global state space can be saved and restored from a binary file using the `load_state` and `save_state` predicates.

The *local state space* is associated with a context and exists for the life of that context. The local state space is normally only accessible to clauses inside the context, thus permitting state information to be stored private to that context.

The unit of storage in state spaces is the Prolog structure with the functor acting as a principal key. (The syntax of structures is described in the chapter titled "Basic Language Elements" in this manual.) For each functor there is an ordered list of structures defining the recall order associated with that functor. The predicates `remember`, `remembera`, `rememberz`, `recall`, `recallz`, `forget` and `forget_all` are used to store structures in a state space, retrieve them, or remove them. These predicates are analogous to the clause space predicates `assert`, `asserta`, `assertz`, `clause`, `retract` and `retractall`.

State spaces are automatically extended when required, memory permitting. Removing items from a state space automatically results in storage being reclaimed. The predicate `new_state` may be used to create and destroy state spaces.

Looped lists (for example, `_X = [a, _X..]`) cannot be stored in state spaces. Also, structures containing intervals or buckets cannot be stored in a state space since they are meaningless when disconnected from their contexts. For similar reasons, constrained variables lose their constraints when stored in state spaces. (These restrictions also apply to storing structures in the clause space.)

State Space Predicates

<code>forget</code>	- removes structures from a state space
<code>forget_all</code>	- removes all structures from a state space
<code>inventory</code>	- generates the principal functors of all structures in a state space
<code>load_state</code>	- loads the global state space
<code>new_state</code>	- creates a new state space
<code>recall</code>	- retrieves structures in recall order
<code>recallz</code>	- retrieves structures in reverse order
<code>remember</code>	- store a structure in a state space at the beginning of the recall order
<code>remembera</code>	- same as <code>remember</code>
<code>rememberz</code>	- store a structure in a state space at the end of the recall order
<code>save_state</code>	- saves the global state space
<code>update</code>	- replaces a structure while maintaining the recall order

forget (_Structure)
forget (_Structure, \$local)

removes structures from a state space

Arguments: forget (+structure)
forget (+structure, +\$local)

Succeeds: forget (_Structure) removes the first structure in the recall order which unifies with _Structure from the global state space. Upon backtracking, any other matching structures are removed. forget (_Structure, \$local) removes the matching structures from the local state space.

If forget succeeds, _Structure will be bound to the structure that was actually removed from the state space.

Fails: forget fails if

- the functor of the structure is a variable
- there are no matching structures, or no state space allocated
- the second argument is not \$local

Note: If forget is placed in the scope of a state space structure generator (for example, [recall(fred(_X)), forget(fred(_X)), fail] it may cause an unexpected termination of the generator.

Examples:

```
/* Store two structures in the state space                                */
?- remember(fred(_X, 3, [_, apple, _Y, _X])).
   ?- remember(fred(_X, 3, [_, apple, _Y, _X])).
YES
?- remember(fred(4, 2, [])).
   ?- remember(fred(4, 2, [])).
YES
```

```
/* Now remove a structure by pattern matching */
?- forget(fred(_X, 3, _U)).
   ?- forget(fred(_X, 3, [_1, apple, _2, _X])).
YES

/* the first matching item has been removed. */
?- recall(fred(_X, _Y, _Z)).
   ?- recall(fred(4, 2, [])).
YES
```

See Also: forget_all in this chapter.

```
forget_all(_Structure)
forget_all(_Structure, $local )
```

removes all structures from a state space

Arguments: forget_all(+structure)
forget_all(+structure, +\$local)

Succeeds: forget_all(_Structure) removes all structures with instantiated functors which unify with _Structure from the global state space . If there are no matching structures forget_all succeeds anyway. forget_all is a filter; _Structure is unchanged. forget_all(_Structure, \$local) removes the matching structures from the local state space.

Fails: forget_all fails if

- the functor of _Structure is a variable
- there are no matching structures, or no state space allocated
- the second argument is not \$local

Examples:

```
/* Store two structures in the state space */
:- remember(fred(_X, 3, [_ , apple, _Y, _X])).
YES
:- remember(fred(4, 2, [ ])).
YES

/* Now remove the structures by pattern matching */
?- forget_all(fred(_X, _Y, _Z)).
?- forget_all(fred(_X, _Y, _Z)).
YES

/* Use recall to verify both structures have been removed */
?- recall(fred(_X, _Y, _Z)).
NO
```

See Also: forget in this chapter.

```
inventory(_Pfunctor)
inventory(_Pfunctor, $local )
```

generates the functors of all structures in a state space

Argument: `inventory(+symbol)`
 `inventory(+symbol, +$local)`

Succeeds: If `_Pfunctor` is a variable, `inventory(_Pfunctor)` generates on backtracking, the principal functors of all structures in the global state space. If `_Pfunctor` is a symbol, then `inventory` verifies the existence of a structure whose principal functor is `_Pfunctor`. `inventory(_Pfunctor, $local)` generates the principal functors of structures in the local state space .

Fails: `inventory` fails if

- `_Pfunctor` is neither a variable, nor a symbol; which is the principal functor of a structure in the state space
- the state space does not exist
- the structure whose principal functor is the symbol `_Pfunctor` does not exist in the state space.
- the second argument is not `$local`

Examples:

```
/* Place some structures in the state space                                        */
:- remember(dog(terrier)),    remember(cat(siamese)).
YES
:- remember(animal(_X) :- furry(_X)).
YES
?- inventory(_Pfunctor).    % now taking inventory...
    ?- inventory('/: -').
    ?- inventory(cat).
    ?- inventory(dog).
YES
```

load_state(_Filename)

loads the global state space

Arguments: load_state(+filename)

Succeeds: Loads the saved global state space from the specified file.
load_state will delete any previously existing global state space.

Fails: load_state fails if _Filename is not the name of a saved state space document (file type APSS), or if there is insufficient memory to load the state space.

Examples:

```
?- load_state('plg:common:example').  
?- load_state('plg:common:example').  
YES
```

See Also: save_state in this chapter.

```
new_state(_Size)
new_state(_Size, $local )
```

creates a new state space

Arguments: new_state(?size)
 new_state(?size, +\$local)

Succeeds: If the argument `_Size` is an integer, a new global state space of `_Size` K bytes is created. If no state space currently exists, `new_state(0)` discards the current global state space. If `_Size` is a variable, `new_state` instantiates `_Size` to the current size of the global state space. `new_state(_Size, $local)` provides equivalent functionality for local state spaces.

Fails: new_state fails if

- `_Size` is less than zero
- `_Size` is greater than zero and a state space already exists
- there is insufficient free memory to create the desired state space
- the second argument is not `$local`

Note: "Remembering" will create a state space with a default size, if no state space exists.

Examples:

```
/* Create a new state space of size 30 Kbytes                   */
?- new_state(30) .
  ?- new_state(30) .
YES
?- new_state(_X) .
  ?- new_state(30) .
YES

/* Discard the current local state space                       */
?- new_state(0, $local) .
  ?- new_state(0, $local) .
YES
```

```
recall(_Structure)
recall(_Structure, $local )
```

retrieves structures from a state space in recall order

Arguments: recall(+structure)
recall(+structure, +\$local)

Succeeds: recall(_Structure) retrieves from the global state space, the first structure in the recall order, which unifies with _Structure. Upon backtracking, any other matching structures are retrieved if they exist. recall(_Structure, \$local) retrieves structures from the local state space.

Fails: recall fails if

- the principal functor of _Structure is a variable
- there are no matching structures or no state space allocated
- the second argument is not \$local

Examples:

```
/* Store some structures in state space. */
:- remember($fred(_X, 3, [_ , apple, _Y, _X]), $local).
YES
:- remember($fred(4, 2, []), $local).
YES

/* Recall the structures */
?- recall($fred(_X, _Y, _Z), $local).
?- recall($fred(4, 2, []), $local).
?- recall($fred(_X, 3 [_1, apple, _2, _X]), $local).
YES

/* Or you can be more specific with the pattern */
?- recall($fred(_X, 2, _Y), $local).
?- recall($fred(4, 2, []), $local).
YES
```

See Also: recallz in this chapter.

```
recallz(_Structure)
recallz(_Structure, $local )
```

retrieves structures from a state space in reverse recall order

Arguments: recallz(+structure)
recallz(+structure, +\$local)

Succeeds: recallz(_Structure) retrieves from the global state space, the first structure in reverse recall order, which unifies with _Structure. Upon backtracking, any other matching structures are retrieved if they exist. recallz(_Structure, \$local) retrieves structures from the local state space.

Fails: recallz fails if

- the principal functor of structure is a variable
- there are no matching structures, or no state space allocated
- the second argument is not \$local

Examples:

```
/* Store some structures in the global state space */
:- remember(fred(_X, 3, [_ , apple, _Y, _X])).
YES
:- remember(fred(4, 2, [])).
YES

/* Recall the structures */
?- recallz(fred(_X, _Y, _Z)).
?- recallz(fred(_X, 3, [_ , apple, _2, _X])).
?- recallz(fred(4, 2, [] )).
YES
```

See Also: recall in this chapter.

```
remember(_Structure)
remembera(_Structure)
remember(_Structure, $local )
remembera(_Structure, $local )
```

stores structure in the global state space at the beginning of the recall order

Arguments: `remember(+structure)`
 `remembera(+structure)`
 `remember(+structure, +$local)`
 `remembera(+structure, +$local)`

Succeeds: `remember(_Structure)` stores the structure, `_Structure`, in the global state space, at the beginning of the recall order for the principal functor of `_Structure`. `remember(_Structure, $local)` stores the structure in the local state space.

`remembera` is the same as `remember`.

Fails: `remember` fails if

- the principal functor of `_Structure` is a variable
- the structure contains intervals, buckets or looped lists
- the second argument is not `$local`

Note: Local names are saved as a reference if they are stored in local state spaces at an equivalent or higher context level (that is, the named objects cannot disappear before the local state space is removed). Global names and local names from higher contexts than the state space level are stored as names, and are re-entered into the symbol table (as necessary) when the terms are recalled from the state space.

Examples:

```
/* Use remember to store structures                                     */
?- remember(sport(swimming, tennis, _Y)).
   ?- remember(sport(swimming, tennis, _Y)).
YES

?- remember(sport(_X, baseball, _Y)).
   ?- remember(sport(_X, baseball, _Y)).
YES

?- remember(sport(football, baseball, soccer)).
   ?- remember(sport(football, baseball, soccer)).
YES

/* Use recall to retrieve the structures                             */
/* Observe the order                                               */
?- recall(sport(_X, _Y, _Z)).
   ?- recall(sport(football, baseball, soccer)).
   ?- recall(sport(_X, baseball, _Z)).
   ?- recall(sport(swimming, tennis, _Z)).
YES
```

See Also: rememberz in this chapter.

```
rememberz ( _Structure )  
rememberz ( _Structure, $local )
```

stores a structure in a state space at the end of the recall order

Arguments: `rememberz(+structure)`
 `rememberz(+structure, +$local)`

Succeeds: `rememberz(_Structure)` stores the structure `_Structure` in a state space at the end of the recall order for the principal functor of `_Structure`.
`rememberz(_Structure, $local)` stores the structure `_Structure` in the local state space.

Fails: `rememberz` fails if

- the principal functor of `_Structure` is a variable
- the structure contains intervals, buckets or looped lists
- the second argument is not `$local`

Note: Local names are saved as a reference if they are stored in local state spaces at an equivalent or higher context level (that is, the named objects cannot disappear before the local state space is removed). Global names and local names from higher contexts than the state space level are stored as names, and are reentered into the symbol table (as necessary) when the terms are recalled from the state space.

Examples:

```
/* Use rememberz to store structures at the bottom of the recall order */
/* bottom of the recall order */
?- rememberz(sport(_X, baseball, _Y)).
   ?- rememberz(sport(_X, baseball, _Y)).
YES

?- rememberz(sport(football, baseball, soccer)).
   ?- rememberz(sport(football, baseball, soccer)).
YES

/* Use recall to retrieve the structures in the order they were stored */
/* Observe the order */
?- recall(sport(_X, _Y, _Z)).
   ?- recall(sport(_X, baseball, _Y)).
   ?- recall(sport(football, baseball, soccer)).
YES
```

See Also: remember and remembera in this chapter.

save_state(_Filename)

saves the global state space

Arguments: save_state(+filename)

Succeeds: Saves the global state space in the file _Filename.

Fails: save_state fails if the file system is unable to write the file.

Examples:

```
/* saves the state in the file "example" of type APSS          */
?- save_state('plg:common:example').
   ?- save_state('plg:common:example').
YES
```

See Also: load_state in this chapter.

```
update(_Structure, _New_structure)
update(_Structure, _New_structure, $local )
```

replaces a structure with a new one while maintaining the recall order

Arguments: update(+structure, +structure)
 update(+structure, +structure, \$local)

Succeeds: update(_Structure, _New_structure) replaces the first occurrence of _Structure in the global state space with _New_structure without changing the order. Upon backtracking, the update will be undone.

 update(_Structure, _New_structure, \$local) replaces the structure in the local state space.

Fails: update fails if

- either _Structure or _New_structure are variables
- the principal functors of _Structure and _New_structure are not instantiated to the same symbol
- there is no matching structure, or the state space does not exist
- the third argument is not \$local

Note 1: Subsequent binding of variables in either structure does not affect the contents of the state space.

Note 2: An atomic transaction composed of multiple updates can be made by
 [update(s1(old1), s1(new1)),
 update(s2(old2), s2(new2)),
 cut].

Note 3: If update is placed in the scope of a generator for the same principal functor, the update structure may be visible to the generator. Unlike forget, it does not unexpectedly terminate the generator.

Examples:

```
/* Use remember to store a structure at the top          */
/* of the state space.                                  */
?- remember(sport(swimming, tennis, _Y)).
   ?- remember(sport(swimming, tennis, _Y)).
YES

/* Use rememberz to store a structure at the            */
/* bottom of the state space                            */
?- rememberz(sport(_X, baseball, _Y)).
   ?- rememberz(sport(_X, baseball, _Y)).
YES

/* Use update to replace the first occurrence of        */
/* the structure                                         */
?- once(update(sport(_A, _B, _C), sport(field, track,
aerobics))).
   ?- once(update(sport(_A, _B, _C), sport(field, track,
aerobics))).
YES

/* Use recall to retrieve the structures                 */
?- recall(sport(_X, _Y, _Z)).
   ?- recall(sport(field, track, aerobics)).
   ?- recall(sport(_X, baseball, _Z)).
YES
```

Chapter 10 Debugger

This chapter describes the facilities provided for debugging BNR Prolog programs.

The Box Model

The debugger is based on the traditional box model of Prolog predicates. The collection of all clauses which define a given predicate is called a predicate definition. In this model, a Prolog predicate is treated as a black box having four ports: `call`, `exit`, `redo` and `fail`. The ports represent the states in which a call may be found during its execution.

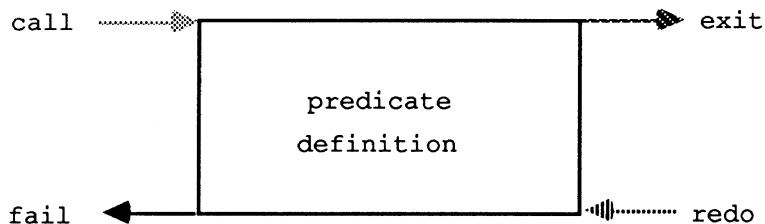


Figure 10-1. The Box Model

The `call` port represents the initial invocation of a predicate. The variable instantiations of the goal displayed in the port message are that of the initial call.

The `exit` port represents the successful completion of a call. That is, the initial goal has successfully unified with the head of some clause and the body of this clause has been satisfied. The variable instantiations of the goal displayed in the port message are that of the return from the call.

The `redo` port represents the situation in which a subsequent goal has failed and the interpreter is backtracking in an attempt to find an alternative solution to a previously satisfied goal. The `redo` indicates that the interpreter is either attempting to resatisfy subgoals in the body of the clause that last succeeded; or, if that fails, is attempting to select a new clause whose head is unifiable with the initial goal and whose subgoals are satisfiable. The variable instantiations of the goal displayed in the port message are that of the previous successful completion of the goal.

The `fail` port represents the failure of the initial goal. The `fail` port may be reached for a variety of reasons including the following: the predicate being called is undefined; the heads of the clauses defining the predicate is not unifiable with the given goal; the subgoals in the bodies of clauses selected cannot be satisfied; the failure of subsequent goals exhaust all solutions to the given goal. The variable instantiations of the goal displayed in the port message are that of the initial call.

The Format of Port Messages

When the debugger displays a port message for a given goal, the message provides information in addition to just displaying the given goal.

The port message is formatted as follows:

```
Status Callid Depth Port: Goal
```

where:

- | | |
|---------------------|---|
| <code>Status</code> | The status of the given goal's principal functor with respect to its being defined, and the predicate defining it being spied. This component of the message occupies the two leftmost character positions. |
| <code>Callid</code> | The unique invocation number for the goal. This component of the message has the form of an integer within parentheses. |
| <code>Depth</code> | The number of direct ancestors of the given goal. This component of the message is an integer. |

Port	The name of the port reached. This component of the message is one of, <code>call</code> , <code>exit</code> , <code>fail</code> , or <code>redo</code> .
Goal	The given goal with its variable bindings reflecting their current instantiation.

A sample port message together with an explanation of the component values follow:

**	(321) 18 redo: p(s(0)) ?
**	The status of the principal functor of the given goal. The ** indicates that the principal functor of the given goal is defined, and the predicate defining it is spied. If the status was white space (that is, two blanks) then this indicates that the principal functor of the given goal is defined, but that the predicate defining it is not spied. If the status was ?? then this indicates that the principal functor of the given goal is not defined.
(321)	The invocation number for the given goal. The <code>call</code> , <code>fail</code> , and <code>exit</code> port messages for this goal have the same number.
18	The number of direct ancestor goals of the given goal. The <code>call</code> , <code>fail</code> , and <code>exit</code> ports messages for this goal have the same number.
redo	The name of the port reached.
p(s(0))	The current instantiation of the given goal.
?	<u>Not</u> part of the message, but rather, it is the prompt indicating that the debugger is waiting for the user to enter a port command.

Interactive Port Commands

When the debugger reaches a leashed port, it outputs the appropriate port message, prompts with a "?", and waits for the user to interactively enter a port command.

Port commands are entered as single keystrokes. An alphabetical list of the available port commands follow:

- a **abort**
Aborts the execution of the top level goal, turns off the debugger, and passes control to the top level of the current invocation of the listener.

- b **break**
Suspends execution and invokes the listener. Essentially, the debugger calls the `break` primitive. When the "break" session is terminated (by typing the `continue` primitive) the debugging session prior to the break is resumed and the last port message of the session is displayed again. Any changes to leashing or spying made during the "break" session remain in effect when the debugging session is resumed.

- c **creep**
Causes the debugger to single step to the next port and output the corresponding port message. If this port is leashed, the user is prompted for another port command. Otherwise, execution of the program continues, possibly outputting other port messages, until either a leashed port is encountered, or the program terminates.

- f **fail**
Fails the current goal and positions control at the point where it is about to backtrack to the next appropriate goal. Issuing the fail command at the fail port is redundant and has no effect.

g ancestor goals
Outputs the direct ancestors of the current goal starting with the root goal and ending with the current goal. The goals output have the following form:

```
Status (N) Depth Goal
```

where:

Status is "***" if the predicate defining the principle functor of Goal is spied, and otherwise is blank.

N is the invocation number of Goal if the execution of Goal was previously reported by the debugger. Otherwise it is "-".

Depth is the number of direct ancestors of Goal.

Goal is a goal.

h help
Displays a reminder of the available port commands.

l leap
Resume execution of the program. The debugger will only output another port message if it encounters a call to a spied or undefined predicate. Otherwise, the execution continues until program termination.

n nodebug
Turns off the debugger and resumes execution of the program.

p print
Outputs the current goal using print.

r	retry	Transfers control back to the point where the current goal was initially called (that is, the <code>call</code> port). Issuing the <code>retry</code> command at the <code>call</code> port is redundant, and has no effect. After issuing a <code>retry</code> , the state of execution is exactly the same as when the call was initially made except for any changes to the state space or clause space caused by <code>remember</code> , <code>forget</code> , <code>retract</code> or <code>assert</code> .
s	skip	Skips from the <code>call</code> or <code>redo</code> port of the current goal to its <code>exit</code> or <code>fail</code> port. Any intervening <code>spy</code> points or calls to undefined predicates are ignored. At the <code>fail</code> and <code>exit</code> ports, the <code>skip</code> port command is equivalent to the <code>creep</code> port command.
?	help	Equivalent to the <code>help</code> port command.
+	spy	If the principal functor of the current goal is defined, a <code>spy</code> point is placed on the predicate defining it.
-	nospy	If a <code>spy</code> point exists on the predicate defining the principal functor of the current goal, it is removed.
<return>	creep	Equivalent to the <code>creep</code> port command.

The Debugger

In keeping with the traditional box model style of the debugger, the BNR Prolog debugger has two start-up modes of operation: "trace" and "debug". The only difference between these two modes concerns the initial predicates for which port messages are output. Once a leashed port is arrived at, the port commands entered determine subsequent debugger modes.

The debugger is turned on in trace mode by means of the `trace` predicate. In this mode the debugger interrupts a program's execution and outputs a port message, immediately upon encountering a goal having a principal functor which is spyable. In

contrast, when the debugger is turned on in debug mode by means of the `debug` predicate, the debugger interrupts a program's execution and outputs a port message, only upon encountering a goal whose principal functor is actually spied. Intuitively, debug mode is like leaping to the first spied predicate, whereas, trace mode is like single-stepping (creeping) a programs execution.

When debugging, certain predicates are of greater interest than others. Such predicates are good candidates for spying. Spy points may be set and removed using the `spy` and `nospyspy` predicates, respectively. When control arrives at a spied predicate, the debugger stops, outputs information, and waits for the user to enter a port command. The use of spy points makes it possible to quickly move through a program, stopping only at the points of interest.

For a more detailed look at a program, creeping (single-stepping) takes the user from the current port to the next port, independent of spy points.

Skipping essentially provides a temporary spy point, disabling preset spy points and creeping.

Leashing specifies the type of ports at which the debugger stops and waits for the user to enter a port command. As well as leashing all predicates, it is possible to leash predicates individually by including a leash list in a spy command

Some predicates are not traceable, and consequently, port messages are not output for these predicates. These predicates are as follows: `attention_handler`, `break`, `capsule`, `continue`, `cut`, `debug`, `fail`, `failexit`, `goal`, `grand_caller`, `is`, `leash`, `listener`, `nodebug`, `nospyspy`, `nospyspyall`, `notrace`, `recovery_unit`, `repeat`, `set_trace`, `spied`, `spy`, `spyall`, `trace`, `tracer`.

The debugger does not make a distinction with respect to a predicate's status of being hidden, nonhidden, closed or open; they are all treated equivalently. Also, for convenience (since local predicates in lower contexts are not normally intended to be seen) they are treated in the following manner: Only those predicates that can be called from the top level of the listener may be spied or crepted.

Debugging information concerning the mode in which the debugger was initialized, and the leashing of nonspied predicates, is retained over context reloads. Spy point information is retained only if the predicates are defined in contexts lower than those being reloaded, otherwise, it must be respecified.

Should control of a program's execution be lost during debugging, then a *Command-*. causes the debugger to immediately enter trace mode with leashing restored to full.

Debugger Predicates

In this section the predicates for turning on and off the debugger, setting and resetting spy points, and leashing ports are discussed. Specifically, the following predicates are discussed:

<code>debug</code>	- turns the debugger on
<code>leash</code>	- sets or queries leashing
<code>nodebug</code>	- turns the debugger off
<code>nospy</code>	- disables spying on specific predicates
<code>nospyall</code>	- disables spying on all spied predicates
<code>notrace</code>	- turns the debugger off
<code>spied</code>	- generates currently spied predicates
<code>spy</code>	- enables spying on specific predicates
<code>spyall</code>	- enables spying on all spyable predicates
<code>trace</code>	- turns the debugger on in trace mode

`debug ()`

turns the debugger on

Arguments: None.

Succeeds: Turns the debugger on and places the debugger in "debug" mode.

Fails: Never fails.

Note 1: Turning the debugger on in debug mode means that the debugger will not interrupt execution until a spied predicate is encountered.

Note 2: Turning the debugger on is independent of the predicates being spied and the ports being leashed.

See Also: `nodebug`, `trace` and `notrace` in this chapter.

leash (*_Ports*)

sets or queries leashing

Arguments: leash(?list)

Succeeds: If *_Ports* is a list consisting of zero or more of the following symbols: *call*, *exit*, *redo*, and *fail*, then *leash* sets the leash list for those predicates not specifically spied. If *_Ports* is a variable then *leash* returns a list of the currently leashed ports.

Fail: *leash* fails if *_Ports* is neither a variable nor a list containing zero or more of the following symbols: *call*, *exit*, *redo*, and *fail*.

Note 1: By default, all four ports are leashed.

Note 2: *leash*([]) removes all leashing. Consequently, the only interaction with the debugger occurs at the leashed ports of the spied predicates.

Note 3: On arrival at a "leashed" port, the debugger outputs the appropriate port message and waits for the user to interactively enter a port command. On arrival at an "unleashed" port, the debugger outputs the appropriate port message, and continues program execution.

Examples:

```
/* leash just the call and exit ports                               */
?-leash([exit, call]).
   ?- leash([exit, call]).
YES
?-leash(_Ports).
   ?- leash([call, exit]).
YES

/* restore leashing to all ports                                   */
:- leash([redo, fail, exit, call]).
YES
```

See Also: spy in this chapter.

`nodebug ()`

turns the debugger off

Arguments: None.

Succeeds: Turns the debugger off.

Fails: Never fails.

Note 1: Turning off the debugger does not remove spy points or alter leashing.

Note 2: `nodebug` and `notrace` are equivalent .

See Also: `debug`, `trace` and `notrace` in this chapter.

`nospy(_P1, _P2, ..., _Pn)`

disables spying on specific predicates

Arguments: `nospy(+symbol, +symbol, ..., +symbol)`

Succeeds: Removes the spy points, if they exist, on the predicates, `_P1`, `_P2`,
`...`, `_Pn`.

Fails: `nospy` fails if

- any of the predicates are not defined
- no predicates are specified (`n=0`)

See Also: spied in this chapter.

`nospyall()`

disables spying on all predicates

- Arguments:** None
- Succeeds:** Disables spying on all spied predicates.
- Fails:** Never fails.
- See Also:** `nospy` in this chapter.

`notrace()`

turns the debugger off

Arguments: None

Succeeds: Turns the debugger off.

Fails: Never fails.

Note 1: Turning off the debugger does not remove spy points or alter leashing.

Note 2: `nodebug` and `notrace` are equivalent .

See Also: `nodebug` in this chapter.

spied(_Predicate)
spied(_Ports, _Predicate)

generates currently spied predicates

Arguments: spied(?symbol)
spied(?list, ?symbol)

Succeeds: If `_Predicate` is a variable then `spied` successively unifies it on backtracking with the predicate names that are currently spied. If `_Predicate` is a symbol then `spied` verifies that `_Predicate` is spied.

In the extended form of `spied`, if `_Ports` is a variable it is unified with the ports of the spied predicate that are leashed. If `_Ports` is a list, it is unified with the list of currently spied ports.

Fails: `spied` fails if

- there are currently no spied predicates
- `_Predicate` is neither a symbol nor a variable
- `_Predicate` is not defined
- `_Predicate` is not defined by a spied predicates
- `_Ports` is not a variable or a list containing zero or more of the following symbols: `call`, `fail`, `exit`, `redo`.

See Also: `spy` in this chapter.

spy(P1, P2, ..., Pn)
spy(Ports, P1, P2, ..., Pn)

enables spying on specified predicates

Arguments: spy(+symbol, +symbol, ..., +symbol)
 spy(+list, +symbol, +symbol, ..., +symbol)

Succeeds: spy places spy points on the the predicates P1, P2, ..., Pn.
 If the extended form of spy is used then only the ports specified by
 the leash list, Ports, are leashed.

Fails: spy fails if

- any of the Pi's are not defined, in which case, none of the predicates are spied.
- Ports is not a list containing zero or more of the following symbols: call, exit, redo and fail.
- no predicates are specified (n=0) .

Note 1: By default, all four ports are leashed.

Note 2: On arrival at a "leashed" port, the debugger outputs the appropriate port message and waits for the user to interactively enter a port command. On arrival at an "unleashed" port, the debugger outputs the appropriate port message, and continues program execution.

See Also: spied and leash in this chapter.

spyall()

enables spying on spyable predicates.

Arguments: None.

Succeeds: Places spy points on all spyable predicates.

Fails: Never fails

See Also: spy in this chapter.

trace

turns debugger on in trace mode

Arguments: None.

Succeeds: Turns the debugger on and places the debugger in "trace" mode.

Fails: Never fails.

Note 1: Turning the debugger on is independent of the predicates being spied and the ports being leashed.

Note 2: Turning the debugger on in trace mode means that the debugger will interrupt the program's execution immediately upon encountering a goal.

See Also: `nodebug`, `debug` and `notrace` in this chapter.

Writing a Debugger

There are three predicates which are fundamental to the BNR Prolog debugging facility, and are available for use when designing and implementing a debugger. They are: `enable_trace`, `set_trace`, and `tracer`.

The predicate `enable_trace` sets, resets, and queries a global trace flag. The format of `enable_trace` is as follows:

```
enable_trace(_FlagSetting)
```

To set the global trace flag, bind `_FlagSetting` to the integer 1. To reset the global trace flag, bind `_FlagSetting` to the integer 0. The current setting of the flag may be queried by leaving `_FlagSetting` as an unbound variable; the variable becomes bound to the setting.

The predicate `set_trace`, sets, resets, and queries a symbol's trace flag. The format of `set_trace` is as follows:

```
set_trace(_Symbol, _FlagSetting)
```

Each symbol known to the system has its own trace flag, and, depending on the status of `_Symbol`, the characteristics of the trace flag are different:

`_Symbol` is a predicate symbol. To set the trace flag, bind `_FlagSetting` to any integer in the range 1 to 255 inclusive. To reset the trace flag, bind `_FlagSetting` to the integer 0, and to query its current value, leave `_FlagSetting` as an unbound variable. The ability to set the trace flag with any number in the range of 1 to 255 presents some interesting possibilities. For example, the BNR Prolog debugging facility uses the value stored in a predicate symbol's trace flag to determine if the predicate is spied, and which of its ports are leashed.

`_Symbol` is a nonpredicate symbol. To set the trace flag, bind `_FlagSetting` to any nonzero integer. To reset the trace flag, bind `_FlagSetting` to the integer 0, and to query its current value, leave `_FlagSetting` as an unbound variable. For such symbols, irrespective of the value used to set the trace flag, querying its trace flag always returns the value -1. Also note that the trace flag for all nonpredicates is initially set.

An exception to the above characteristics of a symbol's trace flag applies: If the symbol in question is currently not a predicate symbol because the clauses defining it have been retracted, and the context from which these clauses have been retracted has not been exited, then, in the transition from a predicate symbol to a nonpredicate symbol, the symbol's trace flag is initialized to have the value 255. Furthermore, the trace flag for such symbols may be reset to have any value in the range 1 to 255 inclusive. However, upon exiting the context, the trace flag for such symbols resorts to those of nonpredicate symbols described above.

The predicate `tracer` is the predefined "catchall" clause. The interpreter, before evaluating any goal having a principle functor, first checks two flags: the global trace flag and the trace flag of the principle functor of the current goal. If both flags are set then the given goal is evaluated (reduced) to the goal `tracer()`. Intuitively, the situation is analogous to having the following clause as the very first clause in the clause space:

```
_P(_Args..) :- [
    enable_trace(_E1),
    (_E1 > 0),
    set_trace(_P, _E2),
    (_E2 > 0),
    cut(_P),
    tracer() ].
```

The immediate ancestor of `tracer` is the goal responsible for the call to `tracer`. The kernel of a debugging facility utilizing the trace flags amounts to an appropriate implementation for the predicate `tracer`.

Predicates for Implementing Debuggers

This section describes a few basic predicates which may be used when designing and implementing a debugger. The predicates, `break`, `continue`, `listener`, `tracer` may be overloaded or replaced entirely by user written Prolog code.

<code>break</code>	– suspends execution
<code>caller</code>	– returns caller's immediate ancestor goal
<code>continue</code>	– resumes execution
<code>enable_trace</code>	– enables and disables tracing
<code>goal</code>	– generates ancestor goals
<code>grand_caller</code>	– returns <code>grand_caller</code> 's callers ancestor goal
<code>listener</code>	– default Prolog system listener
<code>retry</code>	– re-execute a goal
<code>set_trace</code>	– sets the trace flag for symbol
<code>traceback</code>	– outputs the ancestor goal stack
<code>tracer</code>	– calls tracer
<code>try</code>	– executes a goal

break

suspends execution

Arguments: None.

Succeeds: `break` suspends execution and returns the user to interactive mode. The listener prompt is displayed. Execution is resumed by calling the `continue` predicate. When execution is resumed the call to `break` succeeds.

Fails: Never fails.

Notes: A break may be executed from within a break. The break nesting level is output.

caller(_Goal)

returns or verifies the caller's immediate goal

Arguments: caller(?goal)

Succeeds: caller succeeds if _Goal is unifiable with the immediate ancestor of the goal caller(_Goal).

Fails: caller fails if _Goal is not unifiable with the immediate ancestor.

Examples:

```
/* assert a clause                                     */
foo(fred) :- [
    caller(_Goal),
    write('\n***** -> ', _Goal, '\n') ].
OK

?- foo(_X).
***** -> foo(fred).
?- foo(fred).
YES
```

See Also: grand_caller in this chapter.

continue

resumes execution

Arguments: None.

Succeeds: If it exists, the current break session is terminated and the previous session resumed. Otherwise `continue` simply succeeds.

Fails: Never fails.

See Also: `break` in this chapter.

enable_trace(_Setting)

enables and disables tracing

Arguments: enable_trace(?integer)

Succeeds: If _Setting is bound to 0 or 1, or is an unbound variable, enable_trace is used to set, reset and query the global trace flag setting, respectively.

Fails: enable_trace fails if _Setting is not a variable or the integer 0 or 1.

See Also: set_trace in this chapter.

`goal(_Goal)`

generates ancestor goals

Arguments: `goal(?goal)`

Succeeds: On backtracking, successively unifies `_Goal` with those ancestor goals that are unifiable with it.

Fails: `goal` fails if `_Goal` is not unifiable with some ancestor goal.

Examples:

```
/* Assert some clauses                                     */
p(1) :- q(2).
OK

q(2) :- p(3).
OK

p(3) :- q(4).
OK

q(4) :- [nl, goal(_G), write(_G), nl, _G = p(1)].
OK

?- p(X).
q(4)
p(3)
q(2)
p(1)
?- p(1).
YES
```

See Also: `grand_caller` in this chapter.

grand_caller(_Goal)

returns grand_caller's ancestor goal

Arguments: grand_caller(?goal)

Succeeds: grand_caller succeeds if the immediate ancestor of grand_caller's immediate ancestor is unifiable with _Goal.

Fails: grand_caller fails if the immediate ancestor of grand_caller's immediate ancestor is not unifiable with _Goal.

Examples:

```

/* assert a clause defining tracer                                     */
tracer() :- [
    grand_caller(_G),
    write('\n***** -> ', _G),
    nl].
OK

/* assert an arbitrary clause                                       */
foo(_X).
OK

/* set trace flags                                                 */
?- [set_trace(foo, 1), enable_trace(1)].
   ?- [set_trace(foo, 1), enable_trace(1)].
YES

/* call foo                                                         */
?- foo(fred).
***** -> foo(fred).
   ?- foo(fred).
YES

```

See Also: caller in this chapter.

`listener()`

the default Prolog system listener

Arguments: None.

Succeeds: Invokes the system listener.

Fails: `listener` fails if the command read by the invoked listener causes a system error.

retry(_Goal)

re-executes a goal

Arguments: **retry**(+goal)

Succeeds: Re-executes a goal which has an ancestor **try**(_Goal).

Fails: **retry** fails if **try**(_Goal) is not an ancestor on the goal stack.

See Also: **try** in this chapter.

set_trace(_Predicate, _Setting)

sets the trace flag of symbols

Arguments: set_trace(+symbol, ?integer)

Succeeds: Sets, resets and queries the trace flag of symbols.

Fails: set_trace fails if _Setting is not a variable or an integer.

Examples:

```
/* disable the trace flag of foo                                */
?- set_trace(foo, 0).
   ?-set_trace(foo, 0).
YES

/* enable the trace flag of foo                                */
?- set_trace(foo, 1).
   ?-set_trace(foo, 1).
YES
```

`t r a c e b a c k ()`

outputs the goal stack

Arguments: None.

Succeeds: Outputs the top 16 items of the ancestor goal stack to the console.

Fails: Never fails.

try

executes a goal

Arguments: try(+goal)

Succeeds: Executes a goal in a manner permitting a subsequent `retry`.

Fails: try fails if `_Goal` fails

See Also: `retry` in this chapter.

`tracer()`

invokes the tracer

Arguments: None.

Succeeds: This predicate is called whenever the global trace flag and the trace flag on the principal functor of the current goal are both set.

Fails: Never fails.

Chapter 11

System Predicates

System predicates support monitoring and configuration of Prolog memory resources, measuring performance, and accessing date and version information. These predicates are generally operating-system and machine-independent.

Prolog System Predicates

<code>configuration</code>	- configures the Prolog stack sizes and initial predicate
<code>delay</code>	- delays program execution
<code>halt</code>	- exits the Prolog system
<code>memory_status</code>	- queries the stack utilization
<code>quit</code>	- same as <code>halt</code>
<code>restart</code>	- restarts Prolog execution
<code>stats</code>	- initializes performance counters or queries performance measures
<code>timedate</code>	- queries the system time and date
<code>version</code>	- queries the system version number

`configuration(_WS, _GS, _LS, _Initial_goal)`

configures the Prolog stack sizes and initial predicate

Arguments: `configuration(?integer, ?integer, ?integer, ?symbol)`

Succeeds: `configuration` sets or queries the Prolog initial configuration environment using the values specified by the following arguments:

- `_WS` specifies the size of the world stack in Kilobytes.
- `_GS` specifies the size of the global stack in Kilobytes.
- `_LS` specifies the size of the local stack in Kilobytes.
- `_Initial_goal` is a symbol specifying a single executable term to be executed prior to entering normal interactive mode. (Multiple goals can be combined in a list.) Execution is defined by `once(_Initial_goal)`.

These changes come into effect the next time Prolog is launched or restarted (using the `restart` predicate).

If an argument is not instantiated, it is unified with the current setting.

Fails: `configure` fails if

- `_WS`, `_GS` and `_LS` are neither integers nor variables
- `_Initial_goal` is neither a symbol nor a variable

Note 1: A special case exists when the value specified for any of the stack sizes is zero bytes. In this case, when Prolog is either started or restarted, the system chooses values determined by the available free memory for each stack whose value is zero. The system will not initialize unless the available memory is large enough to satisfy certain minimum size criteria.

Note 2: The Prolog development environment has an initial predicate `pde_init`. This predicate is responsible for opening the initial text files (Console and selected files at application launch) and installing the menus.

Examples:

```

/* Query current configuration                                     */
?- configuration(_,_,'_').
   ?- configuration(0, 0, 0, 'pde_init').
YES

/* Set world and global stack sizes to 200 K each, local      */
/* stack to system calculated default, and initial           */
/* predicate to also load context utils                       */
?- configuration(200, 200, 0,
   '[pde_init,load_context(utils)]').
   ?- configuration(200, 200, 0,
   '[pde_init,load_context(utils)]').
YES

/* '?-restart.' or application launch required before        */
/* configuration changes effective                             */

```

See Also: `memory_status` and `build_application` in this chapter.

delay (_Seconds)

delays program execution

Arguments: delay(?number)

Succeeds: Suspends program execution for the specified number of seconds rounded to the nearest tick. (A tick is 1/60 of a second on the Macintosh). If _Seconds is not a number (integer or float) then delay succeeds with a delay of zero.

Fails: delay never fails.

See Also: cputime in the chapter titled "Arithmetic" in this manual.

halt ()

exits the Prolog system

Arguments: None.

Succeeds: `halt` exits the Prolog system. Graph and text windows are closed. The user is prompted to save any active windows whose contents are different from the underlying disk files.

Fails: `halt` never fails. However, it can be aborted and the system restarted, by selecting cancel when prompted to save a text window.

Note: `halt` is the same as `quit`. Both are included to provide compatibility with other Prologs

See Also: `quit` and `restart` in this chapter.

```
memory_status([_WS_List],[_GS_List],[_LS_List],
[_SS_List])
```

queries the stack utilization

Arguments: memory_status(?list, ?list, ?list, ?list)

Succeeds: memory_status unifies its arguments with the current status of the world stack, global stack, local stack and the global state space. Each argument, for example, [_WS_List] is returned in the form of a list of three numbers [_T,_U,_H] which gives the total size, the amount currently in use and the maximum used (the high water mark) since stats() was called. All sizes are expressed in bytes.

Fails: memory_status fails if any of the argument unifications fail.

Note: configuration, described in this chapter, is used to control the sizes of the stacks. new_state described in the chapter titled "State Space Management" is used to set the size of the global state space. The maximum amount of memory used by the state space (that is, the high water mark) will always be zero bytes since the state space is not organized as a stack.

Examples:

```
?- memory_status(_,_,_,_).
?- memory_status([261120, 150112, 159436], [98304,1948,
16196], [52224, 16, 220], [4096,2196,0]).
YES

?- memory_status([261120,_..], [98304,_..], [52224,_..],
[4096,2196,0]).
?- memory_status([261120, 150112, 159436], [98304,1948,
16196], [52224, 16, 220], [4096,2196,0]).
YES
```

See Also: configuration and stats in this chapter.

quit ()

exits the Prolog system

Arguments: None.

Succeeds: quit exits the Prolog system. Graph and text windows are closed. The user is prompted to save any text windows whose contents are different from the underlying text files.

Fails: quit never fails. However, it can be aborted, and the system restarted, by selecting cancel when prompted to save a text window.

Note: quit is the same as halt. Both are included to provide compatibility with other Prologs.

See Also: halt in this chapter.

`restart()`

restarts Prolog execution

Arguments: None.

Succeeds: This predicate restarts execution of the Prolog application according to the current configuration information. All graphics windows are closed; text windows remain open. All user contexts are lost.

Fails: `restart` never fails.

See Also: configuration in this chapter.

```
stats()
stats(_Lis, _Prims, _Intops, _Iterations, _DeltaT)
```

initializes performance counters or queries performance measures

Arguments: None.
 stats(?number, ?number, ?number, ?number, ?number)

Succeeds: stats initializes the following performance counters:

- number of logical inferences
- number of primitive calls
- number of interval operations
- number of narrowing iterations
- time, in ticks (1/60 sec.) since the counters were zeroed

stats also initializes the stack and the state space high water marks.

stats(_Lis, _Prims, _Intops, _Iterations, _DeltaT)
 generates the performance measures listed above.

Fails: stats never fails.

stats(_Lis, _Prims, _Intops, _Iterations, _DeltaT)
 fails if any of the arguments do not unify with the returned values.

Note: Logical Inferences Per Second (LIPS) are usually calculated as
 $(_Lis + _Prims) / (_DeltaT / 60)$.

Examples:

```
?-[stats, [integer_range(_, 1, 10), fail];
stats(,,_,_,_)].
?- [stats, [integer_range(_, 1, 10), fail]; stats(81, 84, 0, 0, 5)]
YES
```

timedate(Time, Date)

queries the system time and date

Arguments: timedate(?symbol, ?symbol)

Succeeds: timedate unifies Time with a symbol containing the system time (in the form 'hh:mm'), and Date to a symbol containing the system date (in the form 'yy mm dd').

Fails: timedate fails if Time and Date do not unify with symbols representing the current time and date.

Examples

```
/* queries the system date and time                                     */
?- timedate(Time, Date).
?- timedate('20:12', '88 02 01').
YES

/* Verifies the system date and time                                   */
?- timedate('20:12', '88 02 01').
?- timedate('20:12', '88 02 01').
YES
```

```
version([_Vints])
```

queries the system version number

Arguments: `version(?list)`

Succeeds: `version` succeeds if `_vints` unifies with a list of integers specifying the Prolog system version number.

Fails: `version` fails if `_vints` does not unify with a list of integers specifying the Prolog system version number.

Examples:

```
/* version used to query the version number                */
?- version(_Vints) .
   ?- version([1,0,0]).
YES
```

Building an Application

An application is a single disk file containing the Prolog run-time system, contexts (including base definitions) and other code required by the application, initial system configuration options, and Macintosh resources. It is intended that the creators of BNR Prolog applications will be able to distribute their applications without restrictions.

BNR Prolog is a special application which supports the program development environment (PDE). Any BNR Prolog application created will not support the full PDE. In particular, the primitives `build_application`, `save_ws`, and `set_trace` will not be available. The actual building of an application is done by using the `build_application` predicate.

Refer to the chapter titled "System Information" in the *BNR Prolog User Guide* for further information on Building Applications.

```
build_application(_Filename, _Signature, _Stack_sizes,  
_Initial_predicate, [_Contexts])
```

builds an application

Arguments: build_application(+filename, +filecreator, ?list,
+goal_list, ?list)

Succeeds: Builds a BNR Prolog application where

_Filename is the pathname of a file to be created

_Signature is a symbol representing the "creator" of the application. The type of the file will be 'APPL'. Signature should be a unique identifier as described in Volume III of *Inside Macintosh*. Signature must be exactly 4 characters in length, padded with blanks if necessary.

_Stack_sizes is a list of three numbers specifying the size of the world stack, global stack, and local stack respectively in Kilobytes. If this argument is unbound, then the current PDE configuration sizes will be used. A stack size of zero causes BNR Prolog to allocate the stack based on the amount of free memory available to the application.

_Initial_predicate is a term specifying the initial goal list. Upon completion of this term, the application exits.

_Contexts is a list of context filenames to be included in the application file. "Current" binary forms will be created if necessary. If uninstantiated or an empty list, no contexts will be added.

Fails:

build_application fails if

- _Filename is an invalid Macintosh file specification, or the file _Filename already exists
- _Signature is not a valid symbol for the creator of the application
- _Stack_sizes is neither a variable nor a list of numbers
- _Initial_predicate is not a term specifying a valid goal list
- _Contexts is neither an variable nor a list of zero or more context filenames

Example:

```
/*To create a simple application that simply brings up a      */
/* dialog with a message and then quits. Try:                */
?- build_application(silly, 'SAMP', _X, 'message(\Hi
there\)', []).

/* If you exit BNR Prolog and launch the program silly,      */
/* you should get a dialog with the words 'Hi there'         */
/* Clicking on OK causes the program to terminate.          */
```

Chapter 12

Macintosh File System Access

A number of predicates are provided for interfacing with the Macintosh hierarchical file system. These predicates are used to copy, delete, and rename files, as well as for generating or testing names of entities in the file system.

Macintosh Filenames, File Types and File Creators

Associated with a file is a filename, a file type and a file creator.

The syntax of valid Macintosh file names is defined in Volume IV of *Inside Macintosh*. In summary, a *full filename* consists of a volume name, followed by a colon (:), followed by zero or more directory (folder) names, each terminated by a colon, and a filename. Each component can be up to 31 characters long, except for volume names which are constrained to 27 characters. *Partial filenames* (also called partial pathnames) omit one or more of the leading components of the full filename and are interpreted relative to the current default directory. Examples of valid filenames are:

```
MyDisk:                % Volume name
MyDisk:MyFolder:MyFile % Full Filename
:MyFolder:MyFile      % Three Partial Pathnames
:MyFile
MyFile
```

A *file type* is a four-character sequence, identifying the type of the file created by the application. The file types used by BNR Prolog are 'TEXT' for a standard text document, 'APWS' for a saved workspace, and 'APSS' for a saved state space. 'APPL', which is the standard application file type, is the file type for the BNR Prolog application.

A *file creator* is a unique four-character sequence which identifies the application which created the file. The file creator for BNR Prolog text documents and work spaces is 'APRO'.

File Predicates

The following is a list of the file predicates:

copyfile	- copies a file
defaultdir	- queries or sets the name of the default directory
deletefile	- deletes a file
fullfilename	- queries or verifies full and partial filenames
homedir	- queries or verifies the name of the home directory
isdirectory	- generates directory names
isfile	- generates file attributes
isvolume	- generates the names of physical disk volumes
listdirectories	- queries the list of directories in the default directory
listfiles	- queries the list of files in the default directory
listvolumes	- queries the list of volumes in the default directory
printfile	- prints a file
renamefile	- renames a file

copyfile(_Filename, _Copyname)

copies a file

Arguments: copyfile(+filename, +filename)

Succeeds: Copies the contents of the file _Filename to the file _Copyname. The previous contents of _Copyname are overwritten.

Fails: copyfile fails if

- _Filename is not the name of an existing file
- _Copyname is not a valid Macintosh file specification
- a file system or memory error occurs

Examples:

```
?- copyfile('MyFile', ':NextFolder:NewFile').
?- copyfile('MyFile', ':NextFolder:NewFile').
YES
```

```
/* Invalid parameter */
?- copyfile('MyFile', _Newfile).
NO
```

defaultdir (_DirectoryName)

queries or sets the name of the default directory

Arguments: defaultdir(?directory_name)

Succeeds: If _DirectoryName is a variable then defaultdir returns the name of the current default directory. If _DirectoryName is a valid Macintosh directory name then this becomes the default directory.

Fails: defaultdir fails if _DirectoryName is neither a variable nor the name of a valid Macintosh directory.

Note: The initial default directory is one from which BNR Prolog was invoked by clicking on a desktop document or application. The default directory name will be affixed to any filename which is not a full filename. This applies to any or all file operations which take a filename.

Examples:

```
/* Querying the default directory                                */
?- defaultdir(_DD).
   ?- defaultdir(':MyDisk:MyFolder:').
YES

/* Setting the default directory                                */
?- defaultdir(':NextFolder:').
   ?- defaultdir(':NextFolder:').
YES
```

See Also: homedir in this chapter.

deletefile (_Filename)

deletes a file

Arguments: delete(+filename)

Succeeds: Deletes the file _Filename. _Filename can be either a full or partial filename.

Fails: deletefile fails if

- _Filename is not the name of an existing file
- the file _Filename is open
- the file _Filename is locked

Examples:

```
?- deletefile('MyDisk:MyFolder:MyFile').
?- deletefile('MyDisk:MyFolder:MyFile').
YES
```

fullfilename(_Partialfilename, _Fullfilename)

queries or verifies full and partial filenames

Arguments: fullfilename(?partialfilename, ?fullfilename)

Succeeds: If the partial pathname is specified, fullfilename returns the corresponding full filename. If the full filename is specified, fullfilename generates the last component of the pathname, which is the actual file name. If both arguments are given, fullfilename succeeds if the partial filename corresponds to the same file as the full filename.

Fails: fullfilename fails if

- either _Fullfilename or _Partialfilename is not the name of an existing file, or a variable
- both arguments are variables

Examples:

```
?- fullfilename('MyFile', _FF).
   ?- fullfilename('MyFile', 'MyDisk:MyFolder:MyFile').
YES

?- fullfilename(_PF, 'MyDisk:OtherFile').
   ?- fullfilename('OtherFile', 'MyDisk:OtherFile').
YES
```

homedir(_Directoryname)

queries or verifies the name of the home directory

Arguments: homedir(?directory_name)

Succeeds: If _Directoryname is a variable, then homedir returns the name of the home (or application) directory. This is the directory which contained the Prolog application file itself, and is not alterable. If _Directoryname is instantiated then homedir succeeds if _Directoryname is the home directory.

Fails: homedir fails if _Directoryname is neither a variable nor the name of the home directory

Examples:

```
/* Query the home directory */
?- homedir(_HD).
?- homedir('MyDisk:Prologdir').
YES

/* Cannot set homedir */
?- homedir('MyDisk:MyFolder').
NO
```

See Also: defaultdir in this chapter.

isdirectory(_DirectoryName)

generates directory names

Arguments: isdirectory(?directory_name)

Succeeds: If _DirectoryName is the name of a directory then isdirectory succeeds if _DirectoryName exists in the default directory. If _DirectoryName is a variable, isdirectory returns the name of a directory in the default directory. On backtracking, other directory names in the default directory will be generated.

Fails: isdirectory fails if

- the _DirectoryName is a not a variable or the name of a directory in the default directory.
- there are no directories in the default directory

Examples:

```
?- isdirectory(_).  
   ?- isdirectory('NextFolder1').  
   ?- isdirectory('NextFolder2').  
YES
```

```

isfile(_Filename, _Filecreator, _Filetype)
isfile(_Filename, _Filecreator, _Filetype, _Datastatus,
_Rsrcstatus, _Created, _Modified)

```

generates file attributes

Arguments: `isfile(?filename, ?filecreator, ?filetype)`

`isfile(?filename, ?filecreator, ?filetype, ?list,
?list, ?list, ?list).`

Succeeds: The shorter form of `isfile` succeeds if a file exists in the default directory with name `_Filename`, creator `_Filecreator`, and type `_Filetype`. If any of the arguments are variables, backtracking will generate all matches to the specified argument pattern.

The extended form adds arguments for the current status of the data and resource fork of the file, as well as for the file creation and last modified dates. The status of each fork is given as a list of two elements: the first element is 1 if the fork is open, 0 if it's closed; the second element gives the current size of the fork in bytes. The creation and last modified dates are lists of the form [Year, Month, Day, Hour, Minute, Second, Weekday].

Fails: `isfile` fails if there are no files in the default directory matching the predicate argument pattern.

Examples:

```

?- isfile(Name, Creator, Type) .
   ?- isfile('MyFile', APRO, TEXT).
   ?- isfile('ApplFile', MYAP, APPL).
   ?- isfile('OtherFile', ?????, TEXT).
YES
?- isfile('MyFile', _, _, _, _, _, _).
   ?- isfile('MyFile', APRO, TEXT, [0,280], [0,0],
[1988,4,22,11,30,12,6], [1988,4,25,16,45,56,2]).
YES

```

isvolume (_Volumename)

generates the names of physical disk volumes

Arguments: isvolume(?volume_name)

Succeeds: If _Volumename is instantiated then isvolume succeeds if the volume _Volumename exists. If _Volumename is a variable then isvolume generates the names of all volumes in the system on backtracking.

Fails: isvolume fails if _Volumename is not a variable or a valid Macintosh volume name for an existing volume.

Examples:

```
?- isvolume(_V).  
   ?- isvolume('MyDisk').  
   ?- isvolume('Floppy1').  
YES
```

listdirectories (_Directorylist)

queries the list of directories in the default directory

Arguments: listdirectories(?list)

Succeeds: Unifies _Directorylist with the list of directory names in the default directory. If there are no directories in the default directory, _Directorylist is unified with the empty list.

Fails: listdirectories never fails.

Examples:

```
?- listdirectories(_L).  
?- listdirectories(['NextFolder1', 'NextFolder2']).  
YES
```

listfiles (_Filelist)

queries the list of files in the default directory

Arguments: listfiles(?filelist)

Succeeds: Unifies _Filelist with the list of filenames in the default directory. If there are no files in the default directory, _Filelist is unified with the empty list.

Fails: listfiles never fails.

Examples:

```
?- listfiles(_F).  
    ?- listfiles(['MyFile', 'ApplFile', 'OtherFile']).  
YES
```

listvolumes (_Volumelist)

queries the list of volumes in the default directory

Arguments: listvolumes(?volumelist)

Succeeds: Unifies _Volumelist with the list of volume names on the system.

Fails: listvolumes never fails.

Examples:

```
?- listvolumes(_L).
   ?- listvolumes(['MyDisk', 'Floppy1']).
YES
```

printfile(_Filename)

prints a file

Arguments: printfile(+filename)

Succeeds: Prints the file _Filename. A dialog is used to control printer operation.

Fails: printfile fails if

- _Filename does not exist
- the user 'cancels' while in the print setup dialog
- the printing manager returns an error code

Examples:

```
?- printfile('MyFile').  
?- printfile('MyFile').  
YES
```

renamefile(_Oldfilename, _Newfilename)

renames a file

Arguments: renamefile(+filename, +filename)

Succeeds: Renames the file _Oldfilename to the name _Newfilename. Any affected window will be renamed.

Fails: renamefile fails if

- _Oldfilename is not the name of an existing file
- the file _Newfilename already exists.

Examples:

```
?- renamefile('MyFile', 'YourFile').
```

```
?- renamefile('MyFile', 'YourFile').
```

```
YES
```

Chapter 13

Windows

Two types of windows are supported by BNR Prolog: text windows (of type `text`) and graphics windows (of type `graf`). Windows of type `text` contain only single font text information and correspond, at least temporarily, to text files. Standard text editing operations can be applied to `text` windows. Most of these functions (scrolling, redrawing, moving) are done automatically by the system in order to support the BNR Prolog desktop. However, they can also be driven by Prolog predicates, in particular, `dotext` and `inqtext`.

Windows of type `graf` are much less restricted than text windows: they may contain graphics objects such as circles and rectangles with various kinds of pen and fill patterns, as well as text in various fonts and sizes. However, users are almost entirely responsible for maintaining the contents of a graphics window. For example, it is the user's responsibility to update the contents of a graphics window when a part of it becomes visible.

This chapter is divided into three sections. The first covers generic window operations, such as the opening, closing, and dragging of both text and graphics windows. The second section describes the predicates for manipulating the contents of text windows, while the third describes the predicates for manipulating the contents of graphics windows. The text and graphics descriptors used with `dotext`, `inqtext`, `dograf` and `inqgraf` predicates are described in the chapters titled "Text Descriptors" and "Graphics Descriptors" in this manual.

Names and Types of Windows

The name and type of a window are often passed as arguments to the predicates described in this chapter. The name of a graphics window can be any symbol. However, all text windows are associated with disk files; when a text window is saved it is written to disk. The name of the text window is the same as that of the disk

file, and the usual naming rules for disk files apply also to text windows. See the chapter titled "Macintosh File System Access" for details on naming files.

Predicates for Handling Windows

This section describes predicates that apply to all types of windows.

<code>activewindow</code>	- queries or sets the currently active window
<code>closewindow</code>	- closes a window
<code>dragwindow</code>	- drags a window
<code>growwindow</code>	- changes the size of a window
<code>hidewindow</code>	- hides a window
<code>iswindow</code>	- generates the relationship between a window, its type and its current visibility status
<code>listwindows</code>	- lists all existing windows
<code>openwindow</code>	- opens a window
<code>positionwindow</code>	- queries or sets the position of a window
<code>sizewindow</code>	- queries or sets the size of a window
<code>zoomwindow</code>	- expands a window

activewindow(_Windowname, _Windowtype)

queries or sets the currently active window

Arguments: activewindow(?symbol, ?symbol)

Succeeds: If the arguments `_Windowname` and `_Windowtype` specify the name and type (`text` or `graf`) of an existing window, then this window becomes the active window. If the window is hidden, it is made visible. If the arguments passed are variables, then they become instantiated to the name and type of the currently active window.

Fails: activewindow fails if

- `_Windowname` is not the name of a window
- `_Windowtype` is not the type of the window `_Windowname`

See Also: hidewindow in this chapter.

closewindow(_Windowname)

closes a window

Arguments: closewindow(+symbol)

Succeeds: Closes the window _Windowname. If the window is a text window, this predicate also closes the associated disk file, but does not write the contents of the window to the disk file before closure.

Fails: closewindow fails if

- _Windowname is not the name of a window
- _Windowname is the name of a text window which was explicitly opened as either a read_window or a read_write_window stream using the open predicate, and the stream has not been explicitly closed

Note: If a text window is opened which does not have a previously existing disk file, and if the contents of the window are not explicitly saved while the window is open, the associated disk file will disappear when the window is closed.

See Also: savetext in this chapter.

```
dragwindow( _Windowname, _Xglobal, _Yglobal,  
[ _Xmin, _Ymin, _Xmax, _Ymax])
```

drags a window

Arguments: dragwindow(+symbol, +integer, +integer, [+integer, +integer, +integer, +integer])

Succeeds: Drags an outline of the window `_Windowname` in response to movements of the mouse. When the mouse button is released the entire window moves to the final drag location, provided this is within the dragging limits of the boundary rectangle. The boundary rectangle is specified by the list consisting of the global coordinates `_Xmin`, `_Ymin`, `_Xmax` and `_Ymax`.

The principal use for this predicate is as a response to a `userdrag` event. When used in this way, the `_Xglobal` and `_Yglobal` coordinate pair should be the location where the mouse was pressed which can be obtained from the data parameters of a `userdrag` event (see the description of `userevent` in the chapter "User Events"). If a regular `usermousedown` event is used to get this drag start location, the coordinates must first be converted to the global coordinate system (see the predicate `localglobal` in the chapter titled "Macintosh System Utility Predicates").

Fails: dragwindow fails if

- `_Windowname` is not the name of a window
- `_Xglobal` and `_Yglobal` are not integers corresponding to valid screen coordinates
- the last argument is not a list of four integers corresponding to valid screen coordinates
- the mouse button is released when the mouse is located outside the boundary rectangle defined by the last argument

Note: It is advisable practice to have the list argument define a boundary rectangle 4 pixels smaller than the screen dimensions. This ensures that at least 4 pixels of a window are always visible as a future dragging handle.

```
growwindow(_Windowname, _Xglobal, _Yglobal,  
[_Minwidth, _Minheight, _Maxwidth, _Maxheight])
```

changes the size of a window

Arguments: growwindow(+symbol, +integer, +integer, [+integer,
+integer, +integer, +integer])

Succeeds: Stretches or shrinks an outline of the window's borders by adjusting the bottom right-hand corner of the window in response to movements of the mouse, until the mouse button is released. When the mouse button is released the window's size is changed to the final grow size. The last list argument is a boundary rectangle defining the maximum and minimum dimensions allowed for adjusting the content portion of the named window.

The principle use for this predicate is as a response to a growwindow event, which is generated only for windows that have been created using documentproc or zoomdocproc window definition parameters (see the description of openwindow in this chapter). The global coordinate pair _Xglobal and _Yglobal define the location of the mouse when the mouse button was pressed. These coordinates can be obtained from the data parameters of a userdrag event (see the description of userevent in the chapter "User Events"). If a regular usermousedown event is used to get this drag start location, the coordinate must first be converted to the global coordinate system. (See the predicate localglobal in the chapter titled "Macintosh System Utility Predicates".)

Fails: growwindow fails if

- _Windowname is not the name of a window
- _Xglobal and _Yglobal are not integers corresponding to valid screen coordinates
- the last argument is not a list of four integers corresponding to valid screen coordinates
- the mouse button is released when the mouse is located outside the boundary rectangle defined by the last argument

Note: Only windows that have been created using `documentproc` or `zoomdocproc` window definition parameters (see the definition of `openwindow` in this chapter) will automatically resize the borders of the window as the mouse is being tracked (a Macintosh Operating System property). However, all windows will adjust their size when the mouse button is released.

See Also: The chapter titled "User Events" in this manual.

hidewindow (*_Windowname*)

hides a window

Arguments: hidewindow(+symbol)

Succeeds: Makes invisible the window *_Windowname*. If that window is currently active, the window underneath it (that is, the window that was last active) is made active.

Fails: hidewindow fails if *_Windowname* is not the name of a window.

See Also: activewindow in this chapter.

iswindow(_Windowname, _Windowtype, _Visible)

generates the relationship between a window, its type and its visibility status

Arguments: iswindow(?symbol, ?symbol, ?symbol)

Succeeds: If all the arguments are instantiated, iswindow succeeds if _Windowname is of _Windowtype (text or graf) and has the specified visibility (visible or hidden). If any of the arguments are variables, iswindow generates all solutions defined by the argument instantiations.

Fails: iswindow fails if

- _Windowname is neither a variable nor the name of a window
- _Windowtype is neither a variable nor a valid type specification
- _Visibility is neither a variable nor one of the two symbols allowed (visible or hidden)
- there is no such relationship between the instantiated arguments

See Also: listwindows in this chapter.

`listwindows(_Windowlist)`

lists windows

Arguments: `listwindows(?list)`

Succeeds: Unifies `_Windowlist` with an ordered list of the names of open windows.

Fails: `listwindows` fails if `_Windowlist` is neither a variable nor a list which unifies with the ordered list of window names.

See Also: `iswindow` in this chapter.

```
openwindow(_Windowtype, _Windowname, pos(_Leftedge,  
_Topedge), size(_Width, _Height), options(_Options..))
```

opens a window

Arguments: openwindow(+symbol, ?filename, pos(?integer,
?integer), size(?integer, ?integer),
options(+variadic))

Succeeds: Opens a window of the type specified by `_Windowtype`. If `_Windowname` is a symbol, the window is given that name. If it is a variable then it is instantiated to 'UntitledX' where X is an integer chosen to ensure uniqueness.

If `_Windowtype` is `text` then a text window is opened containing the contents of the disk file `_Windowname`. If the disk file `_Windowname` does not exist, a disk file is created first and the window on it is opened.

If `_Windowtype` is `graf` then a graphics window is opened. The top-left corner of the content portion of a `graf` window has local coordinates (0,0), the x-axis increases to the right and the y-axis increases down.

The `pos` structure specifies the `_Leftedge` and `_Topedge` values which are pixel offsets from the top-left corner of the Macintosh screen to the top-left corner of the content portion of the window being opened. If either `_Leftedge`, `_Topedge`, or both are variables, a system default will be used and the variables will be instantiated to the default values. The `_width` and `_Height` values of the `size` structure are the actual pixel width and height of the content portion of that window. If either one or both of the `_width` and `_Height` arguments are variables, system defaults are calculated and the variables will be instantiated to those values.

The `options` structure is variadic. The options arguments can include a window definition, `closebox`, `vscroll` (vertical scroll), `hscroll` (horizontal scroll) and `msgbutton`. The last four options are disabled by `noclosebox`, `novscroll`, `nohscroll` and

`nomsgbutton`. The window definition argument may be any one of the Macintosh toolbox defined values (see Volume I and IV of *Inside Macintosh*). These have been made available by the options `documentproc`, `dboxproc`, `plaininbox`, `altdboxproc`, `nogrowdoproc`, `rdocproc`, `zoomdocproc` and `zoomnogrow`.

The default options for text windows are `zoomdocproc`, `closebox`, `vscroll`, `hscroll`, and `msgbutton`. The defaults for graf windows are `rdocproc` and `closebox`. The `vscroll`, `hscroll` and `msgbutton` options for graf windows are not implemented.

Fails:

`openwindow` fails if

- a window of the same name and type is already open
- `_Windowtype` is neither the symbol `text` nor `graf`
- `_Windowname` is neither a symbol nor a variable
- `_Leftedge` and `_Topedge` are neither variables nor integers
- `_Width` and `_Height` are neither variables nor integers
- `_Options..` contains elements that are not supported

Examples:

```
/* this example opens a graf window with a specified name    */
/* position and size, but with default options                */
?- openwindow(graf, 'test graf window', pos(140, 140),
size(200, 200), options()).
?- openwindow(graf, 'test graf window', pos(140, 140),
size(200, 200), options()).
YES
```

```
/* this example opens a text window with default settings    */
/* for name, position & size but without grow or close boxes */
?- openwindow(text, _, pos(_, _), size(_, _),
options(noclosebox, zoomnogrow)).
?- openwindow(text, 'Untitled1', pos(35, 60), size(566, 321),
options(noclosebox, zoomnogrow)).
YES
```

See Also:

`closewindow`, `zoomwindow` and `growwindow` in this chapter.

`positionwindow(_Windowname, _Leftedge, _Topedge)`

queries or sets the position of a window

Arguments: `positionwindow(+symbol, ?integer, ?integer)`

Succeeds: If the arguments `_Leftedge` and `_Topedge` are integers, then `positionwindow` moves the window `_Windowname` without changing its size or shape to the position specified. The `_Leftedge` and `_Topedge` values are the number of pixels offset from the top-left corner of the Macintosh screen to the top-left corner of the content portion of the window being opened. If `_Leftedge` and `_Topedge` are variables then `positionwindow` returns the position of the content portion of the window.

Fails: `positionwindow` fails if

- `_Windowname` is not the name of a window
- `_Leftedge` and `_Topedge` are neither variables nor integers

See Also: `sizewindow` in this chapter.

sizewindow(_Windowname, _Width, _Height)

queries or sets the size of a window

Arguments: sizewindow(+symbol, ?integer, ?integer)

Succeeds: If `_Width` and `_Height` are integers, `sizewindow` expands or shrinks the size of the window `_Windowname` to the width and height specified by `_Width` and `_Height`, which are the actual pixel width and height of the content portion of the window. If either `_Width` or `_Height`, or both are variables, then `sizewindow` returns the the width and height of the window `_Windowname` as specified by the argument.

Fails: `sizewindow` fails if `_Windowname` is not the name of a window.

zoomwindow(_Windowname, _Xglobal, _Yglobal)

expands a window

Arguments: zoomwindow(+symbol, +integer, +integer)

Succeeds: Alternates the size and position of the window `_Windowname` between a user state and a standard state. The standard state is a large window that almost fills the entire screen. The user state is the last window position and size explicitly set by the user (or a program). The `_Xglobal` and `_Yglobal` coordinate pair should be the mouse click location returned from the data arguments of a `userzoom` event (see the description of `userevent` in the chapter "User Events").

Fails: `zoomwindow` fails if `_Windowname` is not the name of a window.

Note: This predicate should only be used in response to `userzoom` events, which in turn are only generated for windows created using `zoomdocproc` or `zoomnogrow` window definition parameters (see the definition for `openwindow` in this chapter).

See Also: The chapter titled "User Events" in this manual.

Manipulating Text Window Contents

Text in a text window is manipulated using a text structure. A text structure alters text in the same way as Apple TextEdit routines, and is used to provide the basic text editing facilities required by an application. This includes selecting, editing and inserting text.

A *text structure* consists of a text descriptor or a list of text descriptors chosen from the available set described in the chapter titled "Text Descriptors" in this manual. Text structures may be either attribute descriptors or output descriptors. *Text attribute descriptors* do not change the content of a text file, for example, the description of a piece of selected text. *Text output descriptors* change the contents of a text file, for example, inserting selected text.

Predicates for Manipulating Text Windows

This section describes the predicates `dotext` and `inqtext` which are used to manipulate text in text windows by means of a text structure argument (either a text descriptor or list of text descriptors). Also described are predicates which test for outstanding changes and discard these changes or save them.

<code>changedtext</code>	- tests for outstanding changes
<code>dotext</code>	- applies text descriptors
<code>inqtext</code>	- inquires about text descriptors
<code>reloadtext</code>	- reloads a window
<code>retargettext</code>	- retargets a text window
<code>savetext</code>	- updates the disk file version of the window

changedtext (`_Windowname`)

tests for outstanding changes

Arguments: `changedtext (+filename)`

Succeeds: `changedtext` succeeds if there are outstanding changes in the window `_Windowname` (that is, changes which have not been written to the associated disk file `_Windowname`).

Fails: `changedtext` fails if

- `_Windowname` is not the name of a text window
- there are no outstanding changes in the window

See Also: `savetext` in this section

`dotext` (`_Windowname`, `_Textstructure`)

applies text descriptors

Arguments: `dotext` (+filename, +textstructure)

Succeeds: `dotext` takes a text descriptor or list of descriptors and applies them to the text window named `_Windowname`. Depending on the descriptor, this could result in text being edited, inserted or selected. Text descriptors in `_Textstructure` that contain variables are ignored.

Fails: `dotext` fails if

- `_Windowname` is not the name of a text window
- `_Textstructure` is a valid text descriptor containing an invalid argument or a list of such descriptors

Note: If a text descriptor or list of descriptors contains only partially instantiated terms or invalid descriptors these descriptors are ignored and only the instantiated descriptors are applied. This feature is particularly useful where descriptors are subject to passive or active constraints (see the chapters on constraints in the *BNR Prolog User Guide*).

Example:

```
/*If the window 'hdisk:prolog:alice' contains only:
without pictures or conversation what is the use of a book
/* the following query selects the first 13 chars, cuts them */
/* into the clipboard and pastes them at the end of the file */
?- dotext('hdisk:prolog:alice',
[selectcabs(0,33), % select the first 33 chars
edit(cut), % cut & put them in the clipboard
selectcabs(end_of_file,end_of_file), % go to eof
edit(paste)]). % paste clipboard there
/* producing
what is the use of a book without pictures or conversation
*/
```

See Also: The chapter titled "Text Descriptors" in this manual.

```
inqtext (_Windowname, _Textstructure)
```

inqtext about text attribute descriptors

Arguments: `inqtext(+symbol, +textstructure)`

Succeeds: `inqtext` unifies the text attribute descriptor or list of descriptors in `_Textstructure` with the appropriate values from the window `_Window`. `inqtext` can only query or verify the attributes of a text window, it cannot change these attributes. These attributes include the number of characters, the number of lines, the current selection positions and the current selection of text (see the chapter titled "Text Descriptors")

Fails: `dotext` fails if

- `_Windowname` is not the name of a text window
- `_Textstructure` is a valid text descriptor containing an invalid argument or list of such descriptors

Examples:

```
/*If the window 'hdisk:prolog:alice' contains only:
"curiouser and curiouser" cried alice
/* and the word 'cried' is the current selection then      */
/* the query                                               */
?- inqtext('hdisk:prolog:alice',
[selectcabs(_Start, _End), % selection positions
selection(_Text), %what is the text
csize(Size)]). % how many characters in the file

/ * produces the answer                                     */
?- inqtext('hdisk:prolog:alice',
[selectcabs(26, 31),selection(cried) ,csize(37)]).
YES
```

See Also: The chapter titled "Text Descriptors" in this manual.

reloadtext (_Windowname)

reloads a window

Arguments: reloadtext (+filename)

Succeeds: Reloads the window `_Windowname` with the contents of the current disk file. Any outstanding changes in the window are lost. If there are no outstanding changes reload succeeds and does nothing.

Fails: reloadtext fails if `_Windowname` is not the name of a text window.

See Also: savetext and changedtext in this chapter.

retargettext (`_Windowname`, `_Newname`)

retargets a text window

Arguments: `retargettext(+filename, +filename)`

Succeeds: Attaches the window `_Windowname` to an existing disk file `_Newname`. After a successful call to `retargettext` the window `_Windowname` is renamed to `_Newname`; the contents of the window are not changed and the file corresponding to `_Windowname` is closed but not saved. The original disk file of `_Newname` is not changed until the window, (now called `_Newname`) is saved.

Fails: `retargettext` fails if

- `_Windowname` is not the name of a text window
- `_Newname` is not a valid file specification for an existing disk file, or is a partial filename and is not in the default directory
- `_Newname` is an open window

Note: If the window `_Windowname` is such that `changedtext(_Windowname)` fails, then, after a successful call to `retargettext`, `changedtext(_Newname)` will also fail. This means that the **File** menu cannot be used to save a window that has been "retargeted". Such a window can be saved only with the `savetext` predicate.

See Also: `changedtext` and `reloadtext` in this chapter.

savetext (`_Windowname`)

updates the disk file version of the window

Arguments: `savetext (+filename)`

Succeeds: Updates the contents of the disk file `_Windowname` to match those of its associated window `_Windowname`. The window remains active.

If there are no outstanding changes in the text, the predicate succeeds and does nothing.

Fails: `savetext` fails if

- `_Windowname` is not the name of a text window
- the disk write fails

See Also: `changedtext` and `reloadtext` in this chapter.

Manipulating Graph Window Contents

Graphical objects in BNR Prolog are displayed in graphics windows of type `graf`. These graphical objects are described in terms of graphics structures which operate within graphics windows.

A *graphics structure* is either a single graphics descriptor or a list of graphics descriptors chosen from the available set described in the chapter titled "Graphics Descriptors" in this manual. Graphics descriptors may be *graphics output descriptors* (for example, drawing a line) or *graphics attribute descriptors* (for example, the thickness of a line).

Predicates for Manipulating Graphics Windows

This section describes the predicates `dograf` and `inqgraf` which are used to manipulate and query graphics windows using a graphics structure argument (either a graphics descriptor or list of graphics descriptors). This section should be read in conjunction with the chapter titled "Graphics Descriptors" in this manual.

<code>dograf</code>	- draws a structure
<code>inqgraf</code>	- returns graphics attribute descriptors

`dograf(_Windowname, _Grafstructure)`

draws a structure

Arguments: `dograf(+symbol, +grafstructure)`

Succeeds: `dograf` takes a graphics descriptor or list of descriptors and applies them to the `graf` window named `_Windowname`. Graphics descriptors containing variables are ignored.

Fails: `dograf` fails if

- `_Windowname` is not the name of a graphics window
- `_Grafstructure` is a valid graphics descriptor containing an invalid argument or a list of descriptors

Note: `_Grafstructure` can consist of nested lists of graphics descriptors, to an implementation dependent limit. If it is a nested list, the graphics descriptors in `_Grafstructure` will be executed in the order determined by a depth-first traversal of the list. The effect of nesting in graphics structures is to preserve the graphics window's attributes set at each level. Each nesting level maintains its own attribute set which is stacked upon further nesting and restored upon return. This provides a convenient tool for localizing attribute changes without querying and remembering all the current attribute settings. For example, in the call

```
dograf(tester, [list_a, [list_b], [list_c], list_d])
```

attribute changes in `list_a` will affect all subsequent drawing, whereas no changes in either `list_b` or `list_c` will affect any drawing in `list_d`, because the attributes which `list_a` left will be restored upon exit from the nested level `b` and `c`. The outermost structure level, whether this is a single descriptor or a list (as in the example), permanently alters the current attribute set.

If a graphics descriptor or list of descriptors contains only partially instantiated terms, these descriptors are ignored and only the instantiated descriptors are applied. This feature is particularly

useful where descriptors are subject to passive or active constraints (See the chapters on constraints in the *BNR Prolog User Guide*.)

Examples:

```
/* first open a graf window */
?- openwindow(graf, test, pos(10, 50), size(400, 300),
options()).
    ?- openwindow(graf, test, pos(10, 50), size(400, 300),
options()).
YES

/* draw a line */
?- dograf(test, [moveabs(75, 75), angle(-45),
line(100)]).
    ?- dograf(test, [moveabs(75, 75), angle(-45), line(100)]).
YES

/* or draw a black rectangle and xor a line through it */
?- dograf(test, [backcolor(black), forecolor(white),
fillpat(clear), rectabs(20, 40, 120, 140),
penmode(xor), moveabs(75, 75), angle(90), line(100)]).
    ?- dograf(test, [backcolor(black), forecolor(white),
fillpat(clear), rectabs(20, 40, 120, 140), penmode(xor),
moveabs(75, 75), angle(90), line(100)]).
YES
```

See Also: The chapter titled "Graphics Descriptors" in this manual

inqgraf(_Windowname, _Attributedescriptor)

returns graphics attribute descriptors

Arguments: inqgraf(+symbol, +attribute_descriptor)

Succeeds: inqgraf unifies the specified list of graphics descriptors with their current values for the window _Windowname.

Fails: inqgraf fails if _Windowname is not the name of a graphics window

Examples:

```
/* the query */
?- inqgraf(test, [pensize(_Width, _Height),
textfont(_Font)]).

/* might return the answer */

?- inqgraf(test, [pensize(1, 1), textfont(3)])
```

See Also: The chapter titled "Graphics Descriptors" in this manual

Chapter 14

Text Descriptors

The text contained in a text window can be examined and transformed by passing a text structure as the second argument to the `dotext` and `inqtext` predicates described in the chapter titled "Windows". A *text structure* consists of a text descriptor or a list of text descriptors.

There are two types of text descriptors:

- text attribute descriptors
- text output descriptors

Text attribute descriptors do not change the contents of the text window and may be used to query the attributes of text using the `inqtext` predicate. They may also be passed as arguments to the `dotext` predicate to modify the attributes of the text in the window without changing its contents.

Text output descriptors, on the other hand, physically change the contents of a text window when they are passed as arguments to the `dotext` predicate. These descriptors cannot be used with `inqtext` for querying.

The position of a character, line, or selection of text in a text window is measured either in terms of the number of characters or lines from the beginning of the file, that is, the *absolute* position, or as a *relative* offset from the current selection position. The absolute position zero is the position of the first character in the file.

A selection of text manipulated by `inqtext` or `dotext` using text descriptors is either translated to a symbol (at most 255 characters long) or treated as a stream. Streams in this context are finite sequences of characters associated with a text file.

Text Attribute Descriptors

The following text attribute descriptors are described in this section.

<code>csize</code>	- queries or sets the number of characters in a text file
<code>lsize</code>	- queries or sets the number of lines in a text file
<code>scandirection</code>	- queries or sets the scan direction
<code>selectcabs</code>	- selects text using absolute character positions
<code>selectcrel</code>	- selects text relative to the current selection using character positions
<code>selection</code>	- returns the current selection or finds the next occurrence
<code>selectlabs</code>	- selects text using absolute line positions
<code>selectlrel</code>	- selects lines of text using line positions relative to the current selection

Descriptions of the text attribute descriptors follow. The success and failure of these descriptors are only relevant when the descriptors are passed as arguments to the `inqtext` or `dotext` predicates, either as a single argument or as a member of a list of descriptors that is passed as an argument. The descriptors themselves cannot be executed directly. All the descriptors operate on windows of type `text`.

csize (`_Size`)

queries or sets the number of characters in a text file

Arguments: `csize` (?integer)

Succeeds: If `csize` is passed as an argument to `inqtext`, `_Size` unifies with the number of characters in the text window. If `_Size` is an integer and `csize` is passed as an argument to `dotext`, it sets the number of characters in the text file to `_Size`.

Fails: `csize` fails if `_Size` is neither a variable nor an integer.

See Also: `lsize` in this chapter.

`lsize(_Size)`

queries or sets the number of lines in a text file

Arguments: `lsize(?integer)`

Succeeds: If `lsize` is passed as an argument to `inqtext`, `_Size` unifies with the number of lines in the text window. If `_Size` is an integer and `csize` is passed as an argument to `dotext`, it sets the number of lines in the text file to `_Size`.

Fails: `lsize` fails if `_Size` is neither a variable nor an integer.

See Also: `csize` in this chapter.

scandirection*queries or sets the scan direction*

Arguments: scandirection(?symbol)**Succeeds:** If `_Direction` is a variable then `scandirection` unifies with the current scanning direction. If `_Direction` is specified (acceptable values are `forward` and `backward`) `scandirection` sets the current scanning direction.**Fails:** `scandirection` fails if `_Direction` is neither a variable nor one of the allowed symbols (`forward` or `backward`).

selectcabs (*_Startchar*, *_Endchar*)

selects text using absolute character positions

Arguments: `selectcabs(?integer, ?integer/end_of_file)`

Succeeds: As an argument to `dotext`, `selectcabs` selects the sequence of characters between `_Startchar` and `_Endchar`, where `_Startchar` and `_Endchar` are absolute character positions, offset from character position zero (the position zero corresponds to the first character in the file). Position values less than zero or greater than the file character size are treated as referring to the beginning and end of the file respectively. The end of file may be specified using the symbol `end_of_file`.

If `selectcabs` is used as an argument to `inqtext`, `_Startchar` and `_Endchar` unify with the absolute character positions of the selection. If no text is selected `_Startchar` equals `_Endchar`.

Fails: `selectcabs` fails if

- `_Startchar` does not unify with an integer
- `_Endchar` does not unify with either an integer or the symbol `end_of_file`
- the value of `_Startchar` is greater than the value of `_Endchar`

Examples:

```
/* Consider a window 'hdisk:prolog:test' consisting of :      */
/* the following text:
The big brown fox jumped over the lazy dog.
/* if the cursor is between j and u in window then          */
/* selection in 'hdisk:prolog:test' is at position 19        */
?- inqtext('hdisk:prolog:test', selectcabs(Start,
End)).
?- inqtext('hdisk:prolog:test', selectcabs(19, 19)).
YES
```

```
/* now select 5 chars ig br between pos 5 and 10 */
?- dotext('hdisk:prolog:test', selectcabs(5, 10)).
   ?- dotext('hdisk:prolog:test', selectcabs(5, 10)).
YES
/* now inquire where the selection is */
?- inqtext('hdisk:prolog:test', selectcabs(Start,
End)).
   ?- inqtext('hdisk:prolog:test', selectcabs(5,10)).
YES

/* now position the cursor at the beginning of the file */
?- dotext('hdisk:prolog:test', selectcabs(0,0)).
   ?- dotext('hdisk:prolog:test', selectcabs(0,0)).
YES
```

See Also:

selectcrel and selectlabs in this chapter.

selectcrel(*_Startchar*, *_Endchar*)

selects text relative to the current selection using character positions

Arguments: `selectrel(?integer, ?integer/end_of_file)`

Succeeds: As an argument to `dotext`, `selectcrel` selects the sequence of characters between `_Startchar` and `_Endchar`, where `_Startchar` and `_Endchar` are relative character positions offset from the current selection. `_Startchar` is the number of characters measured backwards from the beginning of the current selection, and `_Endchar` is the number of characters measured forward from the end of the current selection. Relative position values resulting in absolute positions less than zero or greater than the file character size are treated as referring to the beginning and end of the file respectively. The end of file may be specified using the symbol `end_of_file`. If `selectcrel` is used as an argument to `inqtext`, `_Startchar` and `_Endchar` unify with zero.

Fails: `selectcrel` fails if

- `_Startchar` is not an integer
- `_Endchar` is neither an integer nor the symbol `end_of_file`

Examples:

```
/* Consider a window 'hdisk:prolog:test' consisting of      */
/* the following text:                                     */
The big brown fox jumped over the lazy dog.
/* Position the cursor at the beginning of the file      */
/* with selectcabs and then select brown                  */
?- dotext('hdisk:prolog:test',
[selectcabs(0,0),selectcrel(8, 13)]).
   ?- dotext('hdisk:prolog:test', [selectcabs(0,0),selectcrel(8,
13)]).
YES
```

```
/* now select fox jumped over */
?- dotext('hdisk:prolog:test', selectcrel(6, 16)).
   ?- dotext('hdisk:prolog:test', selectcrel(6, 16)).
YES

/* followed by a query to select the */
?- dotext('hdisk:prolog:test', selectcrel(16, 4)).
   ?- dotext('hdisk:prolog:test', selectcrel(16, 4)).
YES

/* a query of the relative position of the selection */
/* always produces the same answer */
?- inqtext('hdisk:prolog:test', selectcrel(Start,
End)).
   ?- inqtext('hdisk:prolog:test', selectcrel(0,0)).
YES
```

See Also: selectcabs and selectlrel in this chapter.

selection(Text)

returns the current selection or finds the next occurrence

Arguments: selection(?symbol/integer)

Succeeds: If selection is passed as an argument to inqtext, then Text is unified with the symbol formed by the concatenation of the characters in the current selection, provided the current selection is less than or equal to 255 characters. If Text is the stream identifier of a writable stream, the selection will be output to that stream. If the stream is the default input stream, zero, the selection will also be entered to the Prolog system.

If Text is a symbol and selection(Text) is passed as an argument to the dotext predicate, the next occurrence of Text is made the current selection. The position may then be determined by means of one of the select descriptors. This search for Text is case sensitive.

Fails: selection fails if

- selection(Text) is passed to dotext and Text is not instantiated to a symbol
- the current selection is longer than 255 characters, selection(Text) is passed to inqtext and Text is not the stream identifier of a writable stream

Examples:

```
/* If the window 'hdisk:prolog:test' consists of the          */
/* two lines:The first query is '?- write(hello_world).'and  */
/* the second occurrence of it is '?- write(hello_world).'
```

```
/* and the first occurrence of '?- write(hello_world).'
```

```
/* selected, then                                           */
?- inqtext('hdisk:prolog:test', selection(Text)).
   ?- inqtext('hdisk:prolog:test', selection('?-
write(hello_world).')).
YES
```

```
/* if _Text is the stream 0 (console) and the following */
/* query is executed */
?- inqtext('hdisk:prolog:test', selection(0)).
   ?- inqtext('hdisk:prolog:test', selection(0)).
YES
?- write(hello_world).hello_world
   ?- write(hello_world).
YES
/* if the first occurrence of '?- write(hello_world).' */
/* is selected */
/* then this query finds the next occurrence */
?- dotext('hdisk:prolog:test', selection('?-
write(hello_world).')).
```

selectlabs(*_Startline*, *_Endline*)

selects text using absolute line positions

Arguments: `selectlabs(?integer, ?integer/end_of_file)`

Succeeds:

As an argument to `dotext`, `selectlabs` selects the text between `_Startline` and `_Endline`, which are absolute offsets from line position zero (the line position zero corresponds to the line in which first character in the file resides). `_Startline` specifies a number of lines measured backwards from the beginning of the current selection and `_Endline` specifies a number of lines measured forward from the end of the current selection. Position values less than zero or greater than the number of lines in the file are treated as referring to the first and last lines of the file respectively. The last line of the file may be specified using the symbol `end_of_file`. If `_Startline` is greater than `_Endline` the selection is null and begins on `_Startline`.

If `selectabs` is used as an argument to `inqtext`, `_Startline` and `_Endline` unify with the absolute line positions of the current selection. If no text is selected `_Startline` equals `_Endline`.

Fails:

`selectlabs` fails if

- `_Startline` does not unify with an integer
- `_Endline` does not unify with either an integer or the symbol `end_of_file`

Examples:

```
/* If the file 'hdisk:prolog:test' consists of the          */
/* two lines:                                              */
This is the first line
This is the second line
*/
```

```
/* then this query selects the second line */
?- dotext('hdisk:prolog:test', selectlabs(1,2)).
   ?- dotext('hdisk:prolog:test', selectlabs(1,2)).
YES

/* and this one queries which lines are selected */
?- inqtext('hdisk:prolog:test', selectlabs(_X,_Y)).
   ?- inqtext('hdisk:prolog:test', selectlabs(1, 1)).
YES
```

See Also: selectlrel and selectcabs in this chapter.

selectlrel(`_Startline`, `_Endline`)

select lines of text using line positions relative to the current selection

Arguments: `selectlrel`(`?integer`, `?integer/end_of_file`)

Succeeds: As an argument to `dotext`, `selectlrel` selects the section of text between the positions `_Startline` and `_Endline` which are offset relative to the current selection. `_Startline` specifies a number of lines measured backwards from the beginning of the current selection and `_Endline` specifies a number of lines measured forward from the end of the current selection. Relative position values resulting in absolute positions less than zero or greater than the number of lines in the file are treated as referring to the first and last lines of the file respectively. The end of the file may be specified using the symbol `end_of_file`.

Fails: `selectlrel` fails if

- `_Startline` does not unify with an integer
- `_Endline` does not unify with either an integer or the symbol `end_of_file`

See Also: `selectlabs` and `selectcrel` in this chapter.

Text Output Descriptors

The success and failure of the text output descriptors is relevant only when they are passed as arguments to the `dotext` predicate, either as a single argument or as a member of a list of descriptors that is passed as an argument. The descriptors themselves cannot be executed directly. All the descriptors operate on windows of type `text`.

<code>edit</code>	- performs an edit action
<code>insert</code>	- inserts a text string
<code>replace</code>	- replaces text

edit (*_Editaction*)

performs an edit action

Arguments: edit (+symbol)

Succeeds: Performs the action *_Editaction* using the current selection and clipboard contents. *_Editaction* can be one of cut, copy, paste or clear. These use the system scrapbook or clipboard, and provide a means of easily transferring textual data in and out of the Prolog environment.

Fails: edit fails if

- *_Editaction* is not a valid edit action
- the result of a paste produces a window greater than 32 Kilobytes

Examples:

```
/* If the window 'hdisk:prolog:test' is:
The big brown fox jumped over the lazy dog.
*/
/* this query selects the first 13 chars, copies them to      */
/* the clipboard and pastes them at the end of the file      */
?- dotext('hdisk:prolog:test',
  [selectcabs(0,13),edit(copy),
   selectcabs(end_of_file,end_of_file),
   edit(paste)]).
/* producing
The big brown fox jumped over the lazy dog.The big brown
*/
```

See Also: insert and replace in this chapter

insert (*_Text*)

inserts a text string

Arguments: insert (+symbol/integer)

Succeeds: Inserts the text *_Text* at the beginning of the current selection. The current selection is not affected. *_Text* must be a symbol or the stream identifier of a writable stream. If *_Text* is a stream, all printable characters in the stream up to the end of the stream are inserted.

Fails: insert fails if *_Text* is not one of the following: a symbol, a valid stream identifier for a writable stream, a variable or a tail variable.

Note: This descriptor is semantically equivalent to a series of calls to the Macintosh Toolbox TEInsert routine described in Volume I of *Inside Macintosh*.

See Also: edit in this chapter.

replace(*_Text*)

replaces text

Arguments: `replace(+symbol/integer)`

Succeeds: Replaces the current selection with the text `_Text`. `_Text` must be a symbol or the stream identifier of a writable stream. In the latter case, all the text in the stream up to the end of the stream is written to the window. Any characters (including nonprintable characters) may be written to the window. These are interpreted as if they had been entered from the keyboard.

Fails: `replace` fails if `_Text` is not one of the following: a symbol, a valid stream identifier for a writable stream, a variable or a tail variable.

See Also: `edit` and `insert` in this chapter.

Chapter 15

Graphics Descriptors

The contents and attributes of a graphics window can be examined and modified by passing a graphics structure as the second argument to the `inqgraf` and `dograf` predicates described in the chapter titled "Windows".

A *graphics structure* consists of a graphics descriptor or a list of graphics descriptors. Graphics structures are not executed directly but are passed as arguments to either the `inqgraf` or `dograf` predicates which execute them. There are two types of graphics descriptors:

- graphics attribute descriptors
- graphics output descriptors

Graphics attribute descriptors, like text attribute descriptors, do not physically change the contents of the graphics window and can be used to examine the attributes using the `inqgraf` predicate. These descriptors can also be executed by the `dograf` predicate to modify the way a graphics object is subsequently drawn.

Graphics output descriptors, like text output descriptors, physically change the contents of the graphics windows. These descriptors define the graphics structure to be drawn and are executed by the `dograf` predicate. Graphics output descriptors will be ignored by `inqgraf`.

The "absolute" coordinate parameters used by any graphics descriptors such as `position(_X,_Y)` and `rectabs(_X1,_Y1,_X2,_Y2)` are relative to the (0,0) coordinates in the top-left-hand corner of the graphics window. The x-axis increases to the right and the y-axis increases downwards.

The `dograf` and `inqgraf` predicates are described in more detail in the Chapter titled "Windows" in this manual. Further information on using the descriptors is available in the chapter titled "User Interfaces" in the *BNR Prolog User Guide*.

Graphics Attribute Descriptors

Attribute descriptors may be used with either the `inqgraf` or `dograf` predicates. The attributes that these descriptors query or modify always pertain to the window named in the `inqgraf` or `dograf` predicates.

A call to `inqgraf` with an attribute descriptor that contains a variable will succeed by unifying the variables with values that pertain to the graphics window's attributes. If the attribute descriptors are ground terms, a call to `inqgraf` will act as a filter. Since attribute descriptors with `inqgraf` are always used either as queries or filters, the attribute descriptors in this section are explained principally in terms of the action of a fully instantiated descriptor by `dograf`.

A call to `dograf` with an attribute descriptor that contains a variable always succeeds and leaves the variables unbound. This can be useful when such variables are subject to constraints imposed by `freeze` and related predicates (See the chapters titled "Control" and "Passive Constraints" in the *BNR Prolog User Guide* and the chapter titled "Control" in this manual.) On the other hand, if `dograf` is called with an attribute descriptor that is ground, the attributes of the window can be set to new values. Although an attribute change does not have any visible effect on the contents of the window, it may affect how subsequent drawing operations are performed.

One way to modularize the effects of a list of graphics attribute descriptors is for the list to be nested. With a nested list of attribute descriptors, `dograf` only uses the attributes at the top level of the list to cause permanent changes to the window's attributes. All the changes caused by sublists of attribute descriptors are temporary (see the description of `dograf` in the chapter titled "Windows"). A list of sublists of descriptors will be executed in the order determined by a depth-first traversal of the list. Lists of graphics descriptors can be nested to an arbitrary depth.

The following is a list of the attribute descriptors.

angle	- queries or sets the angle of movement of the drawing pen
backcolor	- queries or sets the background color
backpat	- queries or sets the background pattern
fillpat	- queries or sets the fill pattern
forecolor	- queries or sets the foreground color
penmode	- queries or sets the pen transfer mode
penpat	- queries or sets the drawing pen's pixel pattern
pensize	- queries or sets the size of the drawing pen
position	- queries or sets the current drawing pen position
scale	- queries or sets a scale factor
textface	- queries or sets the text face characteristics
textfont	- queries or sets the text font
textmode	- queries or sets the text transfer mode
textsize	- queries or sets the text size
userpat	- loads a pattern resource

angle (`_Degrees`)

queries or sets the angle of movement of the drawing pen

Argument: `angle(?number)`

Succeeds: If `_Degrees` is a number and `angle` is used with `dograf`, it sets the drawing pen's movement angle in degrees. Pen angles are measured positively starting from zero degrees at the 3:00 o'clock position. This attribute is used only by the turtle graphics descriptors.

The initial value is zero degrees.

If `angle` is used with `inqgraf`, then `_Degrees` unifies with the current angle of the drawing pen in the graphics window.

Fails: `angle` fails if `_Degrees` is neither a number nor a variable.

backcolor(*_Color*)*queries or sets background color*

Arguments: backcolor(?integer/symbol)**Succeeds:** If *_Color* is of one of the following color numbers or equivalent symbols

30	=	white,
33	=	black,
205	=	red,
341	=	green,
273	=	cyan,
409	=	blue,
137	=	magenta, or
69	=	yellow

then `backcolor`, used with `dograf`, sets the background color of the graphics window.

The background of a graphics window does not automatically change color when this descriptor is invoked. Rather, all subsequent output descriptors which make use of the background color will use this new value. For example, the shape descriptors, like `rectangle`, use the background color when `clear` is in effect as the current fill pattern.

The initial background color is white.

If `backcolor` is used with `inqgraf` then *_Color* unifies with the current background color of the graphics window.

Fails: `backcolor` fails if *_Color* is not a variable or a valid color number, or one of the predefined symbols listed above.

Note: If you specify a color other than white on a black-and-white output device, it will appear in black.

See Also: `forecolor` in this chapter.

backpat (`_Backpattern`)
backpat (`_PatResId`, `_PatResIndex`)

queries or sets the background pattern

Arguments: `backpat` (?symbol)
`backpat` (?symbol/?integer, ?integer)

Succeeds: If `_Backpattern` is one of the following symbols
white,
lightgray,
gray, or
darkgray

then `backpat`, used with `dograf`, sets the background pattern of the graphics window.

The background of the graphics window does not automatically change pattern when this descriptor is invoked. Rather, all subsequent output descriptors which make use of the background pattern will use this new value. For example, the shape descriptors, like `rectangle`, use the background pattern when `clear` is in effect as the current fill pattern.

If `backpat` (`_Backpattern`) is used with `inqgraf` then `_Backpattern` unifies with the current background pattern of the graphics window.

`backpat` (`_PatResId`, `_PatResIndex`) sets the graphics window's background pattern to the pattern resource specified. The pattern resource may be either of type 'PAT ' or 'PAT#'. A resource of type 'PAT ' loads a single pattern identified by a unique resource ID. A resource of type 'PAT#' is a list of patterns. The list is identified by a unique resource ID. A particular pattern in the list is identified by an index. The arguments are interpreted as follows: If the `_PatResIndex` = 0, then a pattern resource of type 'PAT ' whose pattern resource ID is `_PatResId` is loaded. However, if `_PatResIndex` > 0, then a pattern list resource of type 'PAT#' and

resource ID `_PatResId` is loaded. In this case `_PatResIndex` specifies the pattern in the list to be used.

If `backpat(_PatResId, _PatResIndex)` is used with `inqgraf` then `_PatResId` unifies with the current background pattern of the graphics window. If a resource of type 'PAT ' or 'PAT#' is currently loaded, `_PatResId` unifies with the symbol `resourcepattern` and `_PatResIndex` is ignored.

Fails:

`backpat(_Backpattern)` fails if `_Backpattern` is neither a variable nor a symbol specifying a valid pattern

`backpat(_PatResId, _PatResIndex)` fails if

- `_PatResId` is neither a variable nor an integer specifying a valid pattern resource ID nor a valid pattern symbol
- `_PatResIndex` is neither a variable nor an integer specifying a valid pattern list index
- the pattern resource cannot be loaded

fillpat (*_Fillpattern*)

queries or sets the fill pattern

Arguments: fillpat (?symbol)

Succeeds: fillpat, used with *dograf*, sets the fill pattern in the graphics window. If *_Fillpattern* is one of the following symbols

hollow,
pentype, (that is, the initial pen setting)
usertype, (current penpat or userpat)
clear, or
invert,

then fillpat sets the fill pattern in the graphics window.

The initial fill pattern is pentype.

If fillpat is used with *ingraf* then *_Fillpattern* unifies with the current fill pattern of the graphics window.

Fails: fillpat fails if *_Fillpattern* is neither a variable nor a symbol specifying a valid fill pattern.

Note: The fill pattern is used in conjunction with shape output descriptors. These include the descriptors for rectangles, ovals, rounded rectangles, circles and arcs, regions and polygons.

See Also: penpat and userpat in this chapter.

`forecolor(_Color)`

queries or sets the foreground color

Arguments: `forecolor(?integer/symbol)`

Succeeds: If `_Color` is of the following color numbers or equivalent symbols

30	=	white,
33	=	black,
205	=	red,
341	=	green,
273	=	cyan,
409	=	blue,
137	=	magenta, or
69	=	yellow,

then `forecolor`, used with `dograf`, sets the foreground color of the graphics window.

The foreground color is the color used to render all graphic descriptors unless `clear` is used as either a fill pattern or a pen bit combination mode, in which case the background color is used (see the output descriptor `penmode` in this chapter).

The initial `forecolor` is `black`.

If `forecolor` is used with `inqgraf` then `_Color` unifies with the current foreground color of the graphics window.

Fails: `forecolor` fails if `_Color` is not a variable, or an integer which is a valid color number, or one of the predefined symbols listed above.

See Also: `backcolor` in this chapter.

penmode (`_Transfermode`)

queries or sets the pen transfer mode

Arguments: `penmode` (?symbol)

Succeeds: `penmode`, used with `dograf`, sets the drawing pen's bit combination mode for the graphics window. `_Transfermode` is one of the eight boolean combination functions:

<code>copy</code>	<code>notcopy</code>
<code>or</code>	<code>notor</code>
<code>xor</code>	<code>notxor</code>
<code>clear</code>	<code>notclear</code>

If `penmode` is used with `inqgraf` then `_Transfermode` unifies with the pen's current bit combination mode.

Fails: `penmode` fails if `_Transfermode` is neither a variable nor a valid transfer mode.

Note: The pen mode determines how the pen pattern affects whatever is already in the bit image when it subsequently draws lines and shapes into that image. The penmode does not affect the way text is drawn. (The descriptor `textmode` performs that function). The transfer mode which is passed as an argument to the `penmode` descriptor specifies a boolean operation which determines how the pixels from the source and destination are combined. Each pixel (or bit) in the drawing is paired off with its corresponding destination pixel; the specified Boolean operation is performed, and the result is stored in the destination.

The `copy` operation simply copies each pixel from the source to the destination. The `or`, `xor` and `clear` operations leave destination pixels which correspond to white pixels in the source unchanged, and then perform the appropriate boolean operation on destination pixels which correspond to black pixels in the source. `or` changes those pixels to black, `xor` inverts them and `clear` changes them to white. `notcopy`, `notor`, `notxor` and `notclear` are variants of the operations described above and simply invert each pixel in the

pattern source before performing the basic boolean operations. The eight boolean operations are shown in Figure 15-1.

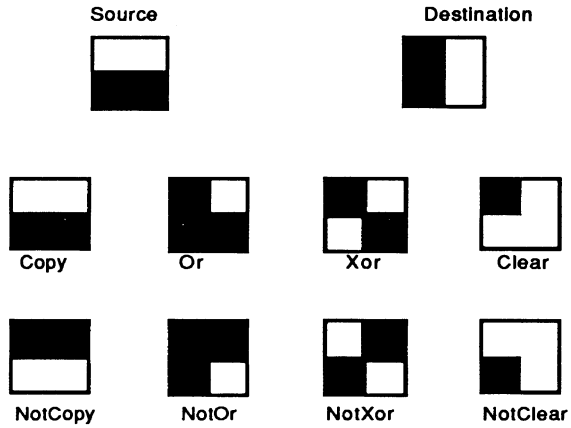


Figure 15 -1 Transfer modes

penpat (*_Penpattern*)
penpat (*_PatResId*, *_PatResIndex*)

queries or sets the drawing pen's pixel pattern

Arguments: penpat (?symbol)
penpat (?integer, ?integer)

Succeeds: If *_Penpattern* is one of the following symbols

white,
lightgray,
darkgray, or
black

then **penpat**, used with **dograf**, sets the drawing pen's pixel pattern in the graphics window.

penpat (*_PatResId*, *_PatResIndex*) sets the drawing pen's pixel pattern in the pattern resource specified by *_PatResId* and *_PatResIndex*. The pattern resource may be either of type 'PAT ' or 'PAT#'. A resource of type 'PAT ' loads a single pattern identified by a unique resource ID. A resource of type 'PAT#' is a list of patterns. The list is identified by a unique resource ID. A particular pattern in the list is identified by an index. The arguments are interpreted as follows: If the *_PatResIndex* = 0 a pattern resource of type 'PAT ' whose pattern resource ID is *_PatResId* is loaded. If, however, *_PatResIndex* > 0 a pattern list resource of type 'PAT#' and resource ID *_PatResId* is loaded. In this case *_PatResIndex* specifies the pattern in the list to be used.

If **penpat** (*_PatResId*, *_PatResIndex*) is used with **inqgraf** then *_PatResId* unifies with the current pen pattern of the graphics window. If a resource of type 'PAT ' or 'PAT#' is currently loaded, *_PatResId* unifies with the symbol resourcepattern and *_PatResIndex* is ignored.

Fails: `penpat` fails if `_Penpattern` is neither a variable nor a symbol specifying a valid pen pattern.

`penpat(_PatResId, _PatResIndex)` fails if

- `_PatResId` is not a variable, or an integer specifying a valid pattern resource ID, or a valid pen pattern symbol
- `_PatResIndex` is neither a variable nor an integer specifying a valid pattern list index
- the pattern resource cannot be loaded

Note: A pattern is a bit image which is 8 pixels wide and 8 pixels high and is used to define a repetitive pattern or tone. These tones are actually progressively denser pixel patterns which result in a progressively darker appearance of the color in use. Thus, `lightgray`, when used while drawing red actually gives light red.

pensize(_Width, _Height)

queries or sets the size of the drawing pen

Arguments: pensize(?integer, ?integer)

Succeeds: If _Width and _Height are integers then pensize, used with dograf, sets the width and height of the drawing pen (measured in pixels). The initial pen size is (_Width = 1, _Height = 1).

If pensize is used with inqgraf then _Width and _Height unify with the current size of the pen.

Fails: pensize fails if _Width and _Height are neither variables nor integers.

Note: If either of the dimensions specified is zero or a negative value, the pen will not draw.

position(_X, _Y)

queries or sets the current drawing pen position

Arguments: position(?number, ?number)

Succeeds: If _X and _Y are instantiated to absolute coordinates in the graphics window, then `position`, used with `dograf`, moves the current drawing pen to that position. The initial value is (_X = 0, _Y = 0).

If `position` is used with `inqgraf` then _X and _Y unify with the current coordinates of the pen.

Fails: `position` fails if _X and _Y are neither variables nor numbers (that is, of type integer or float).

Note: This descriptor is functionally identical to the `moveabs` output descriptor.

scale(_Xscale, _Yscale)

queries or sets a scale factor

Arguments: scale(?number, ?number)

Succeeds: If _Xscale and _Yscale are integers or floats, then *scale*, used with *dograf*, sets a scale factor to be applied to all subsequent positional and size related graphic descriptors. The initial scale factor is (_Xscale = 1.0, _Yscale = 1.0)

If *scale* is used with *inqgraf* then _Xscale and _Yscale unify with the current scale factors of the graphics window.

Fails: *scale* fails if _Xscale and _Yscale are neither numbers (that is, not of type integer or float) nor variables.

textface(*_Styles...*)

queries or sets the text face characteristics

Arguments: textface(?variadic)

Succeeds: Sets the text face characteristics (style) to a sequence of style characteristics specified by the argument *_Styles...*. This sequence can include any combination from the following set:

bold	italic
underline	outline
shadow	condense
extend	

The initial text face is plain (that is, no style or `textface()`).

If `textface` is used with `inqgraf` then the sequence *_Styles...* unifies with the current sequence of text styles of the window.

Fails: `textface` fails if *_Styles...* is neither void nor a sequence (or a partially instantiated sequence) of one or more symbols from the above set, nor a tail variable.

Note: Some of the style characteristics applied to the courier font are shown below:

plain
bold
italic
underline
shadow
condense
extend
underlined italic
bold italic

See Also: `textfont` and `textsize` in this chapter.

textfont (*_Font*)

queries or sets the text font

Arguments: textfont (?integer/symbol)

Succeeds: If *_Font* is one of the following predefined font numbers or equivalent symbols

0	=	systemfont (chicago)
1	=	applfont (geneva)
2	=	newyork
3	=	monaco
4	=	venice
5	=	london
6	=	athens
7	=	sanfran
8	=	toronto
9	=	cairo
11	=	losangeles
20	=	times
21	=	helvetica
22	=	courier
23	=	symbol
24	=	mobile

then `textfont`, used with `dograf`, sets the text font in the graphics window. If the requested font is not available the `applfont` is substituted and will be used by the application unless a font is specified. The initial text font is `applfont`.

If `textfont` is used with `inqgraf` then *_Font* unifies with the symbol corresponding to the current text font for the graphics window.

Fails: `textfont` fails if *_Font* is not a valid font number or a predefined symbol specifying a valid font (see list above), or a variable.

textmode (`_Transfermode`)

queries or sets the text transfer mode

Arguments: `textmode`(?symbol)

Succeeds: `textmode`, used with `dograf`, sets the text transfer mode of the window. `_Transfermode` is one of the following symbols:

`or`
`xor`
`clear`

If `textmode` is used with `inqgraf` then `_Transfermode` unifies with the current text transfer mode of the window.

Fails: `textmode` fails if `_Transfermode` is neither a variable nor a valid text transfer mode symbol.

Note: The text mode determines how text will appear in the graphics window's bit image. The transfer mode which is passed as an argument to the `textmode` descriptor specifies a boolean operation which determines how the pixels from the source and destination are combined. Each pixel (or bit) in the drawing is paired off with the corresponding pixel in the destination; the Boolean operation is performed and the result stored in the destination.

`or` leaves the pixels in the destination unchanged if they correspond to a white pixel in the source. Destination pixels which correspond to black pixels in the source are combined using an `or` operation.

`xor` leaves the pixels in the destination unchanged if they correspond to a white pixel in the source. Black pixels in the source select the destination pixels in the source to be inverted.

`clear` sets every pixel in the destination to white if the corresponding pixel in the source is black.

See also: `textface`, `textfont` and `textsize` in this chapter.

textsize(*_Pointsize*)

queries or sets the text size

Argument: textsize(?integer)

Succeeds: If *_Pointsize* is an integer between one and 127, *textsize*, used with *dograf*, sets the point size of text in the graphics window to that value. The initial text size is the system font size of 12 points which is selected by specifying a size of zero.

If *textsize* is used with *inqgraf* then *_Pointsize* unifies with the current text point size of the graphics window.

Fails: *textsize* fails if *_Pointsize* is neither a variable nor an integer in the range zero to 127 inclusive.

Note: There are 72 points per inch.

userpat (`_PatResId`, `_PatResIndex`)

queries or loads a pattern resource

Arguments: `userpat`(?integer, ?integer)

Succeeds: If `_PatResId` and `_PatResIndex` are integers and `userpat` is used with `dograf`, it loads a pattern resource of type either 'PAT ' or 'PAT#'. A resource of type 'PAT ' loads a single pattern identified by a unique resource ID. A resource of type 'PAT#' is a list of patterns. The list is identified by a unique resource ID. A particular pattern in the list is identified by an index. The arguments are interpreted as follows: If the `_PatResIndex = 0` a pattern resource of type 'PAT ' whose pattern resource ID is `_PatResId` is loaded. If, however, `_PatResIndex > 0` a pattern list resource of type 'PAT#' and resource ID `_PatResId` is loaded. In this case the `_PatResIndex`'th pattern in the list will be used.

If `userpat` is used with `inqgraf` then `_PatResId` and `_PatResIndex` unify with the resource ID and index of the currently loaded pattern resource. If no resource pattern is loaded `_PatResId = 0` and `_PatResIndex = 0`.

Fails: `userpat` fails if

- `_PatResId` is neither a variable nor an integer specifying a valid pattern resource ID
- `_PatResIndex` is neither a variable nor an integer specifying a valid pattern list index
- the pattern resource cannot be loaded

Note: `userpat` does not actually set the pattern, it simply loads the resource file. However, the pattern can subsequently be used as fill pattern for shapes if `usertype` is specified by the `fillpat` descriptor.

See Also: `fillpat` in this chapter.

Graphics Output Descriptors

Graphics output descriptors are passed as arguments to the `dograf` predicate which executes them. As with attribute descriptors, output descriptors containing variables will be ignored by `dograf` and `dograf` will always succeed. If an output descriptor is passed to `inqgraf`, the call to `inqgraf` will fail.

Graphics output descriptors are divided into three general classes:

- `turtle` (supports turtle graphics)
- `relative` (draws relative to the current pen position)
- `absolute` (draws relative to the local origin of the window)

Two other descriptors `polygon` and `region` are used to define arbitrary enclosed spaces and fill them.

Drawing always takes place in the graph window named in the `dograf` predicate.

Numbers may be specified in either integer or floating point format. Numerics include integers, floats and intervals.

Rectangle specifications may be expressed in any order as long as the `x, y, x, y` pairing is maintained.

Turtle Graphics

The descriptors listed below support turtle graphics. The position of the graphics pen is updated after each drawing position.

<code>line</code>	- draws a line
<code>move</code>	- moves the drawing pen
<code>turn</code>	- changes the movement angle of the drawing pen

Relative Output Descriptors

The relative output descriptors draw relative to the current position of the pen.

arcrel	- draws an arc
circlerel	- draws a circle
iconrel	- draws an icon
linere	- draws a line
moverel	- moves the pen
ovalrel	- draws an oval
pictrel	- draws a picture
rectrel	- draws a rectangle
rrectrel	- draws a rounded rectangle
textrel	- draws a text symbol

Absolute Output Descriptors

The absolute descriptors draw relative to the local origin of the window.

arcabs	- draws an arc
circleabs	- draws a circle
iconabs	- draws an icon
lineabs	- draws a line
moveabs	- moves the pen
ovalabs	- draws an oval
pictabs	- draws a picture
rectabs	- draws a rectangle
rrectabs	- draws a rounded rectangle
textabs	- draws a text symbol

Miscellaneous Graphics Descriptors

polygon	- draws an enclosed polygon
region	- draws an enclosed region

Argument Types for the Descriptors

The arguments passed to the graphics descriptors described in this chapter represent different types of graphical data. The argument may specify a point represented by an x and y coordinate or a rectangle represented by its four sides. The coordinates describing the point or rectangle may in turn be relative to the current pen position or absolute with respect to the local origin. They may also be expressed in interval format. Table 15-1 shows the argument types and their variants.

Table 15 -1 Argument types for the graphic descriptors

Argument	Relative	Absolute	Descriptors
point	$\Delta X, \Delta Y$ _Xinterval, _Yinterval	_X, _Y _Xinterval, _Yinterval	circle, line move, text
rectangle	_X1, _Y1, _X2, _Y2 _Xinterval, _Yinterval	_X1, _Y1, _X2, _Y2 _Xinterval, _Yinterval	arc, icon oval, rect rrect, pict


```
arcrel(_ΔX1, _ΔY1, _ΔX2, _ΔY2, _Startangle, _Arcangle)
arcrel(_ΔXi, _ΔYi, _Startangle, _Arcangle)
```

draws an arc (relative)

```
arcabs(_X1, _Y1, _X2, _Y2, _Startangle, _Arcangle)
arcabs(_Xi, _Yi, _Startangle, _Arcangle)
```

draws an arc (absolute)

Arguments: arcrel(+number, +number, +number, +number, +number,
 +number)
 arcrel(+interval, +interval, +number, +number)
 arcabs(+number, +number, +number, +number, +number,
 +number)
 arcabs(+interval, +interval, +number, +number)

Succeeds: Both these descriptors draw an arc of an oval using the current fill pattern. The oval is bounded by the rectangle whose sides are specified. `_Startangle` determines where the arc begins and `_Arcangle` determines its extent relative to `_Startangle`.

The oval defined by `arcrel` is bounded by the rectangle having sides specified by `_X1`, `_Y1`, `_X2` and `_Y2`, which are offsets from the current pen position. The oval drawn by `arcabs` is bounded by the rectangle whose sides are specified by the absolute coordinates `_X1`, `_Y1`, `_X2` and `_Y2`. If interval format is used, the arc is bounded by the rectangle which has x and y axis sides at the boundaries of the intervals specified.

Neither `arcrel` nor `arcabs` change the current pen position.

Fails: `arcrel` fails if

- `_Startangle` and `_Arcangle` are not numbers
- `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2` are not numbers
- `_ΔXi`, `_ΔYi` are not intervals

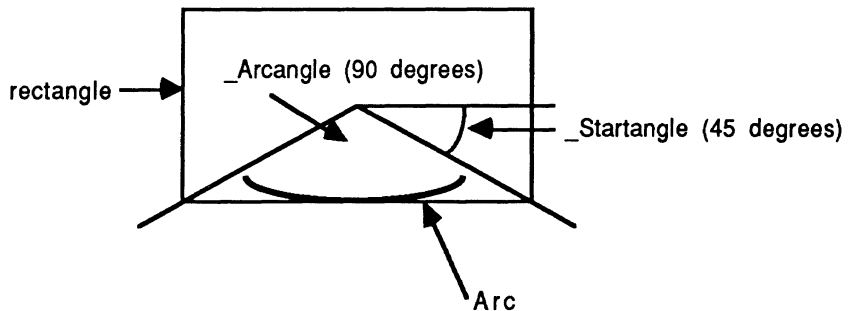
`arcabs` fails if

- `_Startangle` and `_Arcangle` are not numbers
- `_X1`, `_Y1`, `_X2` and `_Y2` are not numbers
- `_Xi`, `_Yi` are not intervals.

Note 1: Angles are measured positively, starting from zero degrees at the 3:00 o'clock position.

Note 2: Angle measures are relative to the enclosing rectangle, that is, a line from the center of the rectangle through the bottom right corner is defined to be at a 45 degree angle even if the rectangle is not a square.

Examples:



```
:- dograf(test, arcrel(10, 80, 110, 10, 45, 90)).
```

```
YES
```

```
/* describes an arc in the graphics window "test" */
```

circrlrel(_ΔXcenter, _ΔYcenter, _Radius)

draws a circle (relative)

circleabs(_Xcenter, _Ycenter, _Radius)

draws a circle (absolute)

Arguments: circrlrel(+numeric, +numeric, +number)
circleabs(+numeric, +numeric, +number)

Succeeds: Draws a circle of given radius _Radius using the current fill pattern.

The center of the circle drawn by circrlrel is specified by the offsets _ΔXcenter and _ΔYcenter which are measured from the current drawing pen position. The center of the circle drawn by circleabs is specified by the absolute coordinates _Xcenter and _Ycenter. If either of the coordinate pairs _ΔXcenter, _ΔYcenter, or _Xcenter, _Ycenter are intervals, the center of the circle will be located at their midpoint.

Neither circrlrel nor circleabs change the current pen position.

Fails: circrlrel and circleabs fail if _Radius is not a number.
circrlrel fails if _ΔXcenter and _ΔYcenter are not numerics.
circleabs fails if _Xcenter and _Ycenter are not numerics.

```
iconrel(_ΔX1, _ΔY1, _ΔX2, _ΔY2, _IconId)
iconrel(_ΔXi, _ΔYi, _IconId)
```

draws an icon (relative)

```
iconabs(_X1, _Y1, _X2, _Y2, _IconId)
iconabs(_Xi, _Yi, _IconId)
```

draws an icon (absolute)

Arguments: iconrel(+number, +number, +number +number, +integer)
iconrel(+interval, +interval, +integer)
iconabs(+number, +number, +number, +number, +integer)
iconabs(+interval, +interval, +integer)

Succeeds: Draws the icon whose resource ID is `_IconId` in the specified rectangle.

The icon drawn by `iconrel` is bounded by the rectangle whose sides are specified by `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2`, which are offset from the current pen position. The icon drawn by `iconabs` is bound by the rectangle whose sides are specified by the absolute coordinates `_X1`, `_Y1`, `_X2` and `_Y2`. If interval format is used, the icon is bounded by the rectangle which has x and y axis sides at the boundaries of the intervals specified by `_ΔXi`, `_ΔYi` (`iconrel`), or `_Xi`, `_Yi` (`iconabs`)

Fails: `iconrel` fails if

- `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2` are not numbers
- `_ΔXi`, `_ΔYi` are not intervals

`iconabs` fails if

- `_X1`, `_Y1`, `_X2` and `_Y2` are not numbers
- `_Xi`, `_Yi` are not intervals

line(_ΔDistance)

draws a line (turtle)

linere1(_ΔX₁, _ΔY₁, ... , _ΔX_n, _ΔY_n)

draws a line (relative)

lineabs(_X₁, _Y₁, ... , _X_n, _Y_n)

draws a line (absolute)

Arguments: line(+numeric)
 linere1(+numeric, +numeric, ..., +numeric, +numeric)
 lineabs(+numeric, +numeric, ..., +numeric, +numeric)

Succeeds: line draws a line as it moves the pen a distance `_ΔDistance` from the current position along a path determined by the current pen angle. If `_ΔDistance` is an interval, the midpoint of the interval is used. `linere1` and `lineabs` draw a line by connecting, in order, each of the specified coordinates. If the coordinates are specified in interval format, then the midpoints of the interval pairs are used. The coordinates specified by the arguments for `linere1` are relative to the current pen position while those for `lineabs` are absolute.

`line`, `linere1` and `lineabs` update the current position of the pen.

Fails: line fails if `_ΔDistance` is not a numeric.

`linere1` and `lineabs` fail if the arguments specified are not numerics.

move(*_Distance*)

moves the drawing pen (turtle)

moverel(*_ΔX*, *_ΔY*)

moves the drawing pen (relative)

moveabs(*_X*, *_Y*)

moves the drawing pen (absolute)

Arguments: move(+number)
 moverel(+numeric, +numeric)
 moveabs(+numeric, +numeric)

Succeeds: move moves the pen along a path determined by the current pen angle. The pen is moved a distance *_Distance* from the current position. If *_Distance* is an interval, the midpoint of the interval is used. moverel moves the pen to a position described by the coordinates *_ΔX*, *_ΔY* measured relative to the current pen position. moveabs performs an absolute movement of the pen to the location specified by the *_X*, *_Y* coordinates. If either of the coordinate pairs *_ΔX*, *_ΔY* or *_X*, *_Y* are specified as intervals, moverel and moveabs move the drawing pen's location to the midpoints of the specified interval pair.

move, moverel and moveabs update the current position of the pen.

Fails: move fails if *_Distance* is not a numeric.
 moverel fails if *_ΔX* and *_ΔY* are not numerics.
 moveabs fails if *_X* and *_Y* are not numerics.

```
ovalrel(_ΔX1, _ΔY1, _ΔX2, _ΔY2)
ovalrel(_ΔXi, _ΔYi)
```

draws an oval (relative)

```
ovalabs(_X1, _Y1, _X2, _Y2)
ovalabs(_Xi, _Yi)
```

draws an oval (absolute)

Arguments: ovalrel(+number, +number, +number, +number)
ovalrel(+interval, +interval)
ovalabs(+number, +number, +number, +numeric)
ovalabs(+interval, +interval)

Succeeds: Draws an oval using the current fill pattern. The oval is bounded by the rectangle whose sides are specified.

The oval drawn by ovalrel is bounded by the rectangle whose sides are specified by _ΔX1, _ΔY1, _ΔX2 and _ΔY2, which are offsets from the current pen position. The oval drawn by ovalabs is bound by the rectangle whose sides are specified by the absolute coordinates _X1, _Y1, _X2 and _Y2. If interval format is used, the oval is bounded by the rectangle which has x and y axis sides at the boundaries of the intervals specified by _ΔXi, _ΔYi (ovalrel) and _Xi, _Yi (ovalabs).

Neither ovalrel nor ovalabs change the current pen position.

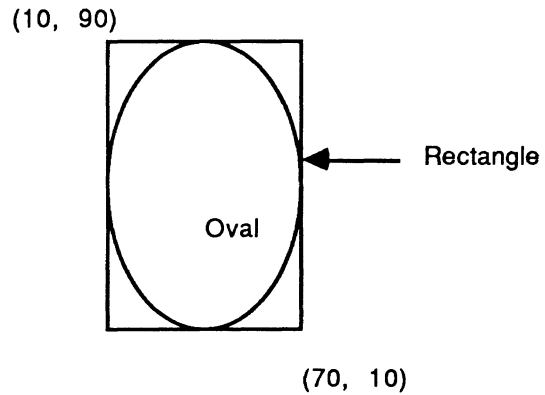
Fails: ovalrel fails if

- _ΔX1, _ΔY1, _ΔX2 and _ΔY2 are not numbers
- _ΔXi and _ΔYi are not intervals

ovalabs fails if

- _X1, _Y1, _X2 and _Y2 are not numbers
- _ΔXi and _ΔYi are not intervals

Examples:



```
:- dograf(test, ovalrel(10, 90, 70, 10) ).  
YES  
/* describes an oval in the graphics window "test" */
```

```
pictrel( _ $\Delta$ X1, _ $\Delta$ Y1, _ $\Delta$ X2, _ $\Delta$ Y2, _PictureID)
pictrel( _ $\Delta$ Xi, _ $\Delta$ Yi, _Picturename)
```

draws a picture (relative)

```
pictabs( _X1, _Y1, _X2, _Y2, _Picturename)
pictabs( _Xi, _Yi)
```

draws a picture (absolute)

Arguments: pictrel(+number, +number, +number, +number, +symbol)
 pictrel(+interval, +interval, +symbol)
 pictabs(+number, +number, +number, +number, +symbol)
 pictabs(+interval, +interval, +symbol)

Succeeds: Both these descriptors draw a picture whose ID is `_PictureID` in the area bounded by the rectangle specified. (This ID is assigned by the `beginpicture` predicate described in this manual in the chapter titled "Pictures".)

`pictrel` draws the picture in the area bounded by the rectangle whose sides are specified by `_ Δ X1`, `_ Δ Y1`, `_ Δ X2` and `_ Δ Y2`, which are offsets from the current pen position. `pictabs` draws the picture in the area bounded by the rectangle whose sides are specified by the absolute coordinates `_X1`, `_Y1`, `_X2` and `_Y2`. If interval format is used, the picture is bounded by the rectangle which has x and y axis sides at the boundaries of the intervals specified by `_ Δ Xi`, `_ Δ Yi` (`pictrel`) or `_Xi`, `_Yi` (`pictabs`).

Fails: `pictrel` and `pictabs` fail if `_Picturename` is not the name of a valid picture.

`pictrel` fails if

- `_ Δ X1`, `_ Δ Y1`, `_ Δ X2`, `_ Δ Y2` are not numbers
- `_ Δ Xi` and `_ Δ Yi` are not intervals

`pictabs` fails if

- `_X1`, `_Y1`, `_X2` and `_Y2` are not numbers
- `_Xi` and `_Yi` are not intervals

`polygon(_Grafstructures..)`

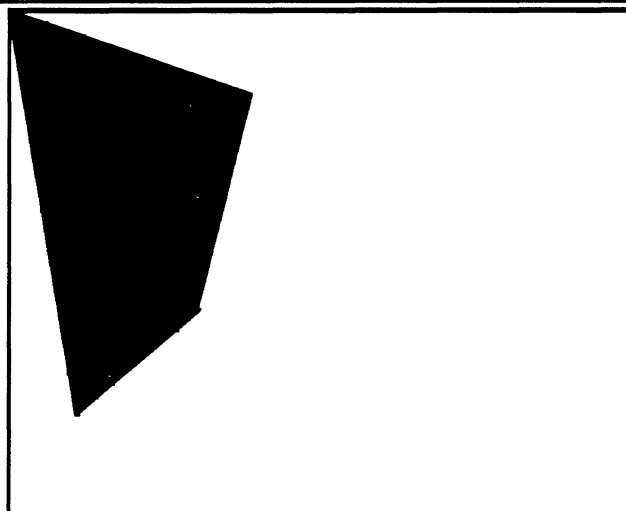
draws an enclosed polygon

Arguments: `polygon(+variadic_grafstructures)`

Succeeds: Draws an enclosed polygon starting from the current pen position in terms of a sequence of graphics descriptors and current fill and pen attributes. Descriptors other than move and line descriptors are ignored.

Fails: `polygon` fails if `_Grafstructures..` is a sequence containing one or more valid graphics descriptors containing invalid parameters.

Examples:



```
/* the call                                                                 */
:- dograf(test, polygon(lineabs(0, 0, 140, 50, 110, 180, 39,
                             240, 0, 0))).
YES
/* draws the polygon above in graf window "test"                            */
```

```
rectrel(_ΔX1, _ΔY1, _ΔX2, _ΔY2)
rectrel(_ΔXi, _ΔYi)
```

draws a rectangle (relative)

```
rectabs(_X1, _Y1, _X2, _Y2)
rectabs(_Xi, _Yi)
```

draws a rectangle (absolute)

Arguments: rectrel(+number, +number, +number, +number)
rectrel(+interval, +interval)
rectabs(+number, +number, +number, +number)
rectabs(+interval, +interval)

Succeeds: Both these descriptors draw a rectangle in the graphics window.

The sides of the rectangle drawn by `rectrel` are specified by `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2`, which are offsets from the current pen position. The sides of the rectangle drawn by `rectabs` are specified by the absolute coordinates `_X1`, `_Y1`, `_X2` and `_Y2`. If interval format is used, the rectangle has x and y axis sides at the boundaries of the intervals specified by either `_ΔXi`, `_ΔYi` (`rectrel`) or by `_Xi`, `_Yi` (`rectabs`).

Neither `rectrel` nor `rectabs` change the current pen position.

Fails: `rectrel` fails if

- `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2` are not numbers
- `_ΔXi` and `_ΔYi` are not intervals

`rectabs` fails if

- `_X1`, `_Y1`, `_X2` and `_Y2` are not numbers
- `_Xi` and `_Yi` are not intervals

```
rrectrel(_ΔX1, _ΔY1, _ΔX2, _ΔY2, _Ovalwidth,  
         _Ovalheight)  
rrectrel(_ΔXi, _ΔYi, _Ovalwidth, _Ovalheight)
```

draws a rounded rectangle (relative)

```
rrectabs(_X1, _Y1, _X2, _Y2, _Ovalwidth, _Ovalheight)  
rrectabs(_Xi, _Yi, _Ovalwidth, _Ovalheight)
```

draws a rounded rectangle (absolute)

Arguments: rrectrel(+number, +number, +number, +number, +number,
+number)
rrectrel(+interval, +interval, +number, +number)
rrectabs(+number, +number, +number, +number,
+number, +number)
rrectabs(+interval, +interval, +number, +number)

Succeeds: Both these descriptors draws a rectangle with rounded corners using the current fill pattern. The sides of the rectangle are specified and the curvature of the corners is determined by an oval having the specified dimensions.

The sides of the rectangle drawn by rrectrel are specified by _ΔX1, _ΔY1, _ΔX2 and _ΔY2, which are offsets from the current pen position. The sides of the rectangle drawn by rrectabs are specified by the absolute coordinates _X1, _Y1, _X2 and _Y2. If interval format is used, the rectangle has x and y axis sides at the boundaries of the intervals specified by _ΔXi, _ΔYi (rrectrel) or _Xi, _Yi (rrectabs).

Neither rrectrel nor rrectabs change the current pen position.

Fails: `rrectrel` and `rrectabs` fail if `_Ovalwidth` and `_Ovalheight` exceed the width and height of the rectangle

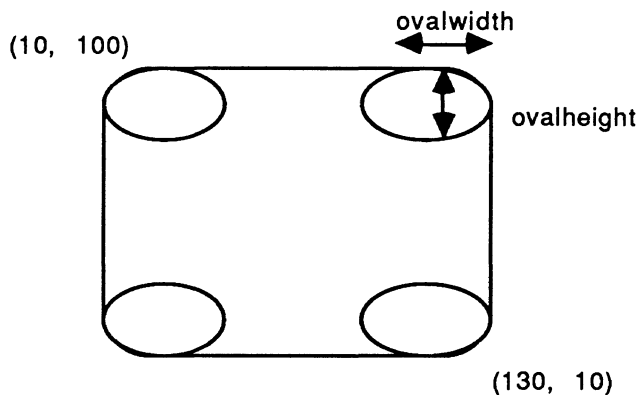
`rrectrel` fails if

- `_ΔX1`, `_ΔY1`, `_ΔX2` and `_ΔY2` are not numbers
- `_ΔXi` and `_ΔYi` are not intervals

`rrectabs` fails if

- `_X1`, `_Y1`, `_X2` and `_Y2` are not numbers
- `_Xi` and `_Yi` are not intervals

Examples:



Rounded Rectangle

```
:- dograf(test, rrectabs(10, 100, 130, 10, 15, 20))
YES
/* describes a rounded rectangle in the graphics window "test" */
```

region(_Grafstructures..)

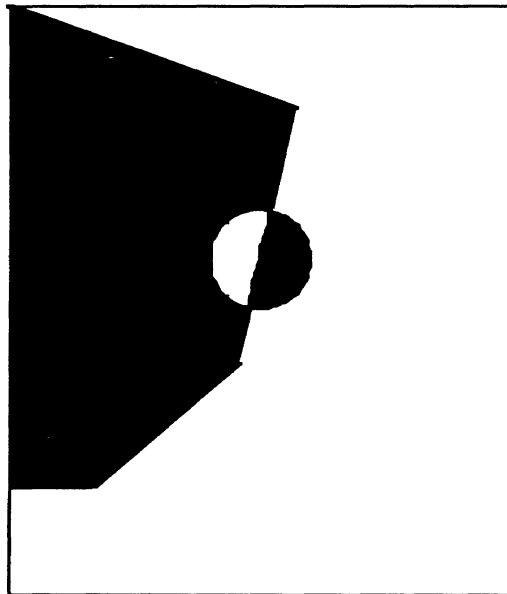
draws an enclosed region

Arguments: region(+variadic_grafstructures)

Succeeds: Draws an enclosed region from the current pen position using a sequence of graphics descriptors, and current fill and pen attributes. Arc descriptors are ignored.

Fails: region fails if _Grafstructures.. is a sequence containing one or more valid graphics descriptors containing invalid arguments.

Examples:



```
/* the call */
:- dograf(test, region(lineabs(0,0,140, 50, 110, 180,
39, 240, 0, 240, 0, 0), ovalabs(100, 100,150, 150))).
/* draws the above region in the graphics window "test" */
```

textrel(*_DeltaXpos*, *_DeltaYpos*, *_Symbol*)

draws a text symbol (relative)

textabs(*_Xpos*, *_Ypos*, *_Symbol*)

draws a text symbol (absolute)

Arguments: textrel(+numeric, +numeric, +symbol)
textabs(+numeric, +numeric, +symbol)

Succeeds: Draws the text symbol *_Symbol* using the current text attributes.

The starting point of the text drawn by `textrel` is specified by *_DeltaXpos* and *_DeltaYpos* which are offsets from the current pen position. The starting point of the text drawn by `textabs` is specified by the absolute coordinates *_Xpos* and *_Ypos*. If interval format is used, the starting point of the text is at the midpoint of the interval pair specified. The current position is updated to reflect the rightmost edge of the text as it is drawn. The text is drawn horizontally and to the right.

`textrel` and `textabs` update the current position of the pen.

Fails: `textrel` and `textabs` fail if *_Symbol* is not of type symbol.

`textrel` fails if *_DeltaXpos* and *_DeltaYpos* are not numerics

`textabs` fails if *_Xpos* and *_Ypos* are not numerics.

`turn(_Δdegrees)`

changes the movement angle of the drawing pen

Arguments: `turn(+number)`

Succeeds: Changes the drawing pen's movement angle by an amount `_Δdegrees` relative to its current value.

Fails: `turn` fails if `_Δdegrees` is not a number.

Note 1: Pen angles are measured positively from zero degrees at the 3:00 o'clock position.

Note 2: This attribute value is referenced only by the turtle graphic output descriptors.

See Also: `angle` in this chapter.

Chapter 16

Pictures

About Quickdraw Pictures

The predicates documented in this chapter allow the creation and manipulation of Quickdraw pictures. Pictures are sequences of drawing commands which can be saved and replayed later with a single predicate call. This provides an easy way to transmit graphical information between Macintosh applications.

Internally, a picture is assigned an integer identifier (ID) (usually, when a call is made to the `beginpicture` predicate) which identifies the picture. Pictures may also be stored as 'PICT' resources in the resource fork of a file. Resources are identified by type, and either an ID or a name. 'PICT' is a standard Macintosh resource type for Quickdraw pictures. Users are referred to Volume I of *Inside Macintosh* for further information on resources and Quickdraw.

Creating and Manipulating Pictures

To create a picture, first define it by making a call to the `beginpicture` predicate. This begins the definition of a picture resource in memory and assigns a picture ID to it. When a picture is defined, the rectangle that surrounds it is specified. This rectangle is called the frame of the picture, and it defines the boundaries of the picture in local coordinates. The local origin (0,0) is the top-left corner of the window's boundary rectangle. The contents of the picture are then created by calls to the `dograf` predicate and its associated descriptors which are described in the chapters titled "Windows" and "Graphics Descriptors" in this manual. Those calls do not produce any visible results. Instead, a picture structure is created which can be drawn later (by executing the picture graphics descriptors `picrel` or `picabs` by means of subsequent calls to `dograf`) or exported to another application by means of the clipboard or scrapbook. Once the picture has been defined, the creation of the picture resource in memory is

terminated by making a call to the `endpicture` predicate. The picture can then be saved using the predicate `savepicture` and played back using `loadpicture`. Memory used by the picture resource can be reclaimed using `deletepicture`. Two other predicates, `attachpicture` and `detachpicture`, logically connect and disconnect a picture to a specified graphics window. Once a picture is attached to a graphics window, any updates for the window are handled by the operating system; no update events are generated.

Refer to the chapter titled "User Interfaces" in the *BNR Prolog User Guide* for further information on using pictures.

Predicates for Manipulating Pictures

<code>attachpicture</code>	- attaches a picture to a graphics window
<code>beginpicture</code>	- begins the creation of a picture
<code>deletepicture</code>	- deletes a picture from memory
<code>detachpicture</code>	- detaches a picture from a graphics window
<code>endpicture</code>	- ends the creation of a picture
<code>ispicture</code>	- defines a relation between a picture, a window and its boundaries
<code>listpictures</code>	- lists defined pictures
<code>loadpicture</code>	- loads a picture from a file
<code>picttoscrap</code>	- writes a picture to scrap
<code>savepicture</code>	- saves a picture
<code>scraptopict</code>	- loads a picture from scrap

attachpicture (*_PictureId*, *_Windowname*)

attaches a picture to a graphics window

Arguments: attachpicture(+integer, +symbol)

Succeeds: Logically connects the picture identified by the picture ID *_PictureID* to the graphics window *_Windowname*. Updates for this graphics window, occurring subsequent to this attachment, are handled by the operating system.

Fails: attachpicture fails if

- *_PictureId* is not a valid picture ID
- *_Windowname* is not the name of a graphics window
- the graphics window *_Windowname* is already attached to a picture

Note: Only one picture can be attached to a graphics window at any time.

See Also: beginpicture and detachpicture in this chapter.

beginpicture(_Windowname, frame(_Top, _Left, _Bottom, _Right),
_PictureId)

begins the creation of a picture

Arguments: beginpicture(+symbol, frame(+integer, +integer,
+integer, +integer), -integer)

Succeeds: Creates a picture resource in memory that has the current attributes of the graphics window `_Windowname`. The structure, `frame`, defines the boundaries of the picture in local coordinates. A new picture ID is created and bound to `_PictureId`. The contents of the picture may then be created by `dograf` calls. Those calls do not produce visible results, but will create the picture structure which can later be drawn into the graphics window using the `dograf` graphics descriptors `pictrel` or `pictabs`.

Fails: `beginpicture` fails if

- `_Windowname` is not the name of a graphics window
- `_Top`, `_Bottom`, `_Right` and `_Left` are not integers that correspond to window coordinates
- `_PictureId` is not a variable

See Also: `endpicture` in this chapter.

deletepicture (*_PictureId*)

deletes picture from memory

Arguments: deletepicture(+integer)

Succeeds: Deletes the picture structure identified by *_PictureId* and its associated contents.

Fails: deletepicture fails if *_PictureId* is not a valid picture ID.

Note: Delete a picture resource from memory only when you have finished with it and need to reclaim the space in memory. Use *savepicture* to save the picture on disk prior to deleting it.

See Also: *savepicture* in this chapter.

detachpicture (*_PictureId*)

disconnects a picture from a graphics window

Arguments: detachpicture(+integer)

Succeeds: Logically disconnects the picture identified by the picture ID *_PictureId* from the graphics window to which it was previously attached. The operating system will no longer update the window as required; update events will be generated.

Fails: detachpicture fails if *_PictureId* is not a valid picture ID or the picture is not attached to a window.

See Also: attachpicture in this chapter.

endpicture (_PictureId)

ends the creation of a picture

Arguments: endpicture (+integer)

Succeeds: End the creation of the picture identified by the picture ID
 _PictureId.

Fails: endpicture fails if _PictureId is not a valid picture ID.

See Also: beginpicture in this chapter.

ispicture(_PictureId, _Windowname, frame(_Left, _Top, _Right, _Bottom))

defines a relation between a picture, a window and its boundaries

Arguments: ispicture(?integer, ?filename, frame(?integer, ?integer, ?integer, ?integer))

Succeeds: ispicture succeeds if a relationship exists between the arguments specified. If one or more of the arguments are variables, ispicture generates values which define the relation.

Fails: ispicture fails if

- _PictureId is neither a variable nor a valid picture ID
- _Windowname is neither a variable nor the name of a graphics window
- _Left _Top, _Right, _Bottom are neither variables nor integers representing valid window coordinates
- a relationship does not exist between the arguments specified

listpictures (Picturelist)

lists defined pictures

Arguments: listpictures(?list)

Succeeds: Unifies Picturelist with the list of defined picture ID.

Fails: listpictures fails if Picturelist does not unify with the ordered list of picture IDs.

```
loadpicture(_Filename, _ResoureId, _PictureId, frame(_Top,  
_Left, _Bottom, _Right))
```

loads the picture from a file

Arguments: loadpicture(+filename, +integer, -integer,
frame(?integer, ?integer, ?integer, ?integer))

Succeeds: Loads the picture resource with resource ID `_ResourceId` from the resource fork of the file `_Filename` and instantiates `_PictureId` with its picture ID making it available for use. The structure `frame` defines the boundaries, in local coordinates, in which the picture was originally drawn.

Fails: loadpicture fails if

- `_Filename` is not a valid file specification for an existing file, or is a partial filename and is not in the default directory
- `_ResourceId` is not a valid 'PICT' resource ID
- `_PictureId` is not a variable
- `_Top`, `_Left`, `_Bottom`, and `_Right` do not unify with the picture's frame dimensions

See Also: savepicture in this chapter.

picttoscrap(_PictureId)

writes a picture to scrap

Arguments: picttoscrap(+integer)

Succeeds: Write the picture specified by _PictureId to the clipboard.

Fails: picttoscrap fails if _PictureId is not a valid picture ID.

See Also: scryptopict in this chapter.

savepicture(_Filename, _ResourceId, _Pictname, _PictureId)

saves a picture

Arguments: savepicture(+filename, +integer, +symbol, +integer)

Succeeds: Writes the picture identified by _PictureId as a 'PICT' resource, with resource ID _ResourceId to the resource fork of the file _Filename. The resource is assigned the name _Pictname. (This name is not meaningful to the Prolog system, but may be significant to other utilities such as resedit). If a 'PICT' resource with same resource ID already exists, it will be overwritten.

Fails: savepicture fails if

- _Filename is not a valid Macintosh file specification
- _ResourceId is not a valid resource ID
- _Pictname is not a symbol
- a disk write error occurs

See Also: loadpicture in this chapter.

scraptopict (_PictureId)

loads a picture from scrap

Arguments: scraptopict (-integer)

Succeeds: Loads a picture from the clipboard and assigns it an ID of
 _PictureID.

Fails: scraptopict fails if _PictureId does not unify with the assigned
 ID.

Chapter 17

Menus

This chapter describes the predicates provided to support the creation, installation, manipulation and deletion of programmer-defined and system-supplied menus.

The menu bar, which always appears at the top of the Macintosh screen, contains the titles of all menus associated with the current application. Each application has its own set of titles. When BNR Prolog is loaded, the menu bar contains the set of menus associated with the BNR Prolog system. These include the **Apple**, **File**, **Edit**, **Find**, **Window** and **Context** menus. Each menu consists of a vertical list of menu items displayed inside a rectangle (See Figure 17-1).

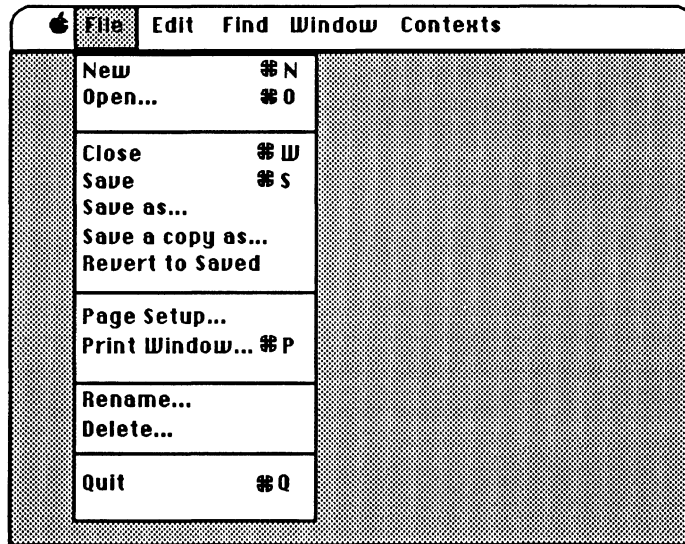


Figure 17-1 File menu of the BNR Prolog desktop

Each of the menu items may have any of the following attributes:

- An icon on the item's left which is a symbolic representation of the item.
- A mark, either a check mark or some other character which denotes the status of an item.
- The command-key sequence used to invoke the command from the keyboard.
- A character style such as italic, bold or underline.
- A dimmed appearance indicating that the item is disabled and cannot be selected by the user.

The menu items of the **File**, **Edit** and **Find** menus in the BNR Prolog system may not be deleted or have their text changed. However, their attributes may be changed and new items may be added to, or deleted from, the end of these menus. The system supplied version of the **Window** menu may not be altered at all.

All the menus can be deleted and replaced by user-defined menus.

The addition of new menus should be accompanied by rules for the `menuselect` predicate that define the actions for that menu. (See the chapter titled "Menus" in the *BNR Prolog User Guide* for examples and further details.)

Predicates for Handling Menus

There are two methods for creating a menu in BNR Prolog. One method is to add a menu to the menu bar with `addmenu` and then to add menu items with `additem`. Every menu created in this way must be given a title and a unique identifying number, its menu ID. This ID number is used both to add items to the menu and to place other menus relative to it. Each menu item has an associated name and item ID (its position from the top), as well as a symbol that defines its attributes, such as its associated keyboard command sequence. A special item ID is the symbol `end_of_menu` which refers to the ID of the last item in the menu. Items can also be added to a menu from resources with the predicate `addresitems`.

A menu can be built by creating a menu resource in the resource fork of the application file using a software development environment such as *Macintosh Programmer's Workshop (MPW)*.

These kinds of menus can be installed with the predicate `addresmenu`. The `addresmenu` predicate takes a resource ID argument which identifies the menu resource to be installed. The menu resource ID should be known in advance.

The symbols 'Apple', 'File', 'Edit', 'Find', 'Window' and 'Contexts' can be used to refer to the system menu IDs or the resource IDs in all the menu predicates that expect these arguments. Use of these symbols explicitly designates the system versions of the menus, and not programmer provided versions, even if the name and contents of the menus are identical. Items within a menu are fully specified by the menu ID and the item ID. For all the menu predicates the resource ID, menu ID and Item ID arguments must be valid, that is, they must be integers corresponding to valid resources, menus and items respectively.

BNR Prolog also supports hierarchical menus and popup menus. Refer to Volume V of *Inside Macintosh* for more information on these types of menus.

The predicates discussed in this chapter are:

<code>additem</code>	- adds an item to an existing menu
<code>addmenu</code>	- installs a menu
<code>addresitems</code>	- adds an item from a resource file
<code>addresmenu</code>	- adds a menu from resource file
<code>deleteitem</code>	- deletes a menu item
<code>deletemenu</code>	- deletes a menu
<code>lastmenudata</code>	- queries the menu ID and item ID of the last menu selection
<code>menuitem</code>	- sets or queries a menu item
<code>popupmenu</code>	- displays a popup menu

additem(*_Item*, *_Attributes*, *_MenuId*, *_After_itemId*, *_ItemId*)

adds a menu item to an existing menu

Arguments: additem(+symbol, +symbol, +integer, +integer/end_of_menu, ?integer)

Succeeds: Adds a menu item having the name *_Item* with display attributes *_Attributes* to the existing menu *_MenuId* after the item *_ItemId*. A value of zero for *_After_itemId* places the new item at the top of the menu before any existing items. The predefined symbol *end_of_menu* may be used for *_After_itemId* to place the new item at the end of the menu after any existing menu items.

_ItemId is unified with the ID of the new item. This value will be one greater than *_After_itemId* if the insertion point is a specified integer, but it may be a new piece of information if the item was added to the end of a menu using the *end_of_menu* symbol. The item ID is required to subsequently manipulate menu items using the predicates *deleteitem* and *menuitem*.

The special character hyphen (-) may be used as item to create a dividing line across the full width of the menu. Typically, this dividing line is also disabled using the attribute specifications described below. An item attribute specification is a symbol containing the menu metacharacters: ^, !, <, /, (and). Attributes accumulate within an *additem* call and are applied in order. The absence of a particular attribute specification results in the default value for that attribute.

^<char>
adds the icon having a resource ID equal to the ASCII value of *<char>* to the item. (Note that this limits menu icons to those having resource IDs <= 255.) The default icon value is none.

!<char>
marks the item with *<char>* on the left of the menu item name. The default value is none.

<<char>

where <char> is one of P, B, I, O, U or S sets the character style to plain, bold, italic, underline or shadow respectively. Attributes of this kind may be used multiple times to specify more than one character style for a particular menu item (for example, '<B<I<S'). The value P (plain) signifies no style attributes, canceling any previously defined style attributes. The default character style is plain.

/<char>

associates <char> with the item, allowing the item to be invoked from the keyboard as a command-key sequence. The default command key equivalent is none.

(

is a single character metacharacter. It disables the item.

)

is also a single character metacharacter. It enables the item. Since menu items are default enabled, this metacharacter is not required for `additem`. However, it is useful in the `menuitem` predicate to enable a disabled menu item.

Fails:

`additem` fails if

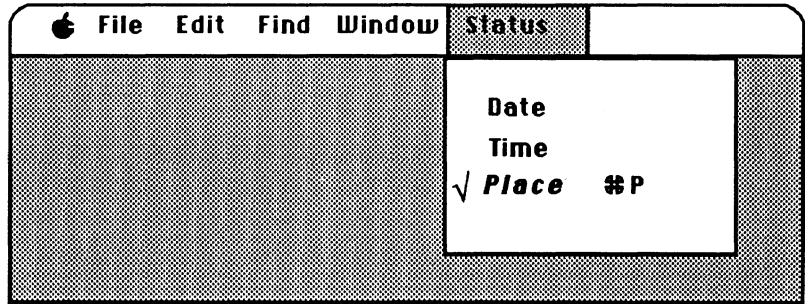
- `_Item` is not a symbol
- `_Attributes` is not a symbol constructed out of one or more of the menu metacharacters
- `_MenuId` is not a valid menu ID number or system menu symbol
- `_After_itemId` is neither the integer zero, the symbol `end_of_menu` nor a valid item ID for an item in the menu specified.

Note:

The characters semicolon (;) and return, while defined as metacharacters in *Inside Macintosh*, are not supported.

Examples:

```
/* adds the item "Place" in bold italics to the end of the */
/* menu Status (whose menu ID is 1). Using the command-key */
/* sequence %P is equivalent to selecting 'Place' from the */
/* menu */
?- additem('Place', '!√<B<I/P', 1, end_of_menu,
  _ItemId).
?- additem('Place', '!√<B<I/P', 1, end_of_menu, 3).
```



See Also: addressitems in this chapter.

```
addmenu(_MenuId, _Title, _Before_menuId)
```

adds a menu

Arguments: addmenu(+integer, +symbol, +integer)

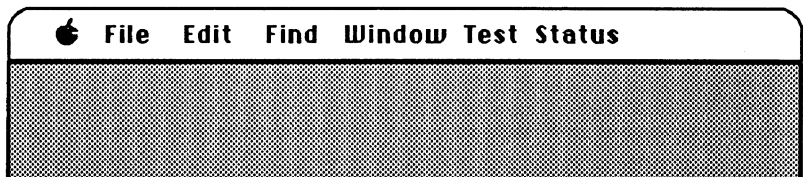
Succeeds: Installs a menu having the name `_Title` into the menu bar ahead of the existing menu `_Before_menuId`. `_MenuId` must be instantiated to an integer which is the new menu ID. If `_Before_menuId` is zero the new menu is placed at the right-hand end of any existing menus in the menu bar. Hierarchical and popup menus are created (but not displayed) if `_Before_menuId` has value of -1 (minus one).

Fails: addmenu fails if

- `_MenuId` or `_Title` are not symbols
- `_Before_menuId` is neither zero nor -1, nor an integer corresponding to a valid menu ID, nor a system menu symbol

Examples:

```
/* Installs a new menu Test (whose menu ID is 2 before      */
/* the menu Status (whose menu ID is 1)                    */
:- addmenu(2, 'Test', 1).
YES
```



See Also: addressmenu in this chapter. For more information on popup and hierarchical menus see Volume V of *Inside Macintosh*.

addresitems(*_Resource_type*, *_MenuId*, *_After_itemId*)

adds an item from a resource file


Arguments: addresitems(+symbol, +integer, +integer)

Succeeds: Adds the names of all resources of type *_Resource_type*, found in all open resource forks, to the existing menu *_MenuId* after the item *_After_itemId*. The resource type is a four character symbol corresponding to a Macintosh resource type. A value of zero for *_After_itemId* places the new items at the top of the menu before any existing items. The predefined symbol *end_of_menu* may be used for *_After_itemId* to place the new items at the end of the menu.

Fails: addresitems fails if

- *_Resource_type* is not a valid resource type
- *_MenuId* is not an integer corresponding to a valid menu ID number
- *_After_itemId* is not the integer zero, the symbol *end_of_menu* or the valid ID for an item in the menu specified

Note 1: Resource types are case sensitive.

Note 2: addresitems is used to install the names of all the desk accessories in the  menu using the resource type 'DRVR'. It can also be used to build a font selection menu using a resource type of 'FONT' or a picture selection menu using the resource type 'PICT'.

Examples:

```
/* Installs all the 'FONT' resources in the system file      */
/* to the user-defined menu 'Test' whose menu ID is 1, to  */
/* the bottom of the menu.                                  */
:- addresitems('FONT',1,end_of_menu )
YES
```

See Also: additem in this chapter.

addressmenu (ResourceId, Before_menuId)

adds a menu from a resource file

Arguments: addressmenu(+integer/+symbol, +integer)

Succeeds: Installs a menu from a resource in the application file's resource fork. The resource is identified by the resource ID ResourceId. The menu will be placed in front of the previously installed menu whose menu ID is Before_menuId. If Before_menuId is zero, the new menu is placed at the right-hand end of any existing menus in the menu bar.

Fails: addressmenu fails if

- ResourceId is neither an integer corresponding to a valid resource ID number nor a system menu symbol
- Before_menuId is not an integer corresponding to valid menu ID number

Note: Menu resources can be created using a software development environment like MPW. The system provided versions of the **File**, **Edit** and **Find** menus exist as resources and may be used as templates for making custom menu resources using a resource editor such as resedit in the MPW. The resource ID of an edited system menu resource must be changed before it is customized.

Examples:

```

/* Installs a new menu Edit before the menu File.          */
/* Has the effect of interchanging the file and edit menus.   */
:- addressmenu(32130, 'File').
YES
/* The same effect is achieved by the question                */
:- addressmenu('Edit', 'File').
YES

```

See Also: addmenu in this chapter.

deleteitem(_MenuId, _ItemId)

deletes a menu item

Arguments: deleteitem(+integer, +integer)

Succeeds: Deletes the item `_ItemId` from the existing menu `_MenuId`. The predefined symbol `end_of_menu` may be used as `_ItemId` to identify the last item in the menu.

Fails deleteitem fails if

- `_MenuId` is not an integer identifying an existing, nonsystem menu
- `_ItemId` is neither an integer identifying an item in the menu identified by `_MenuId` nor the symbol `end_of_file`.

Note: The menu ID and item ID are obtained when calls are made to the predicates `addmenu` and `additem` respectively.


deletemenu (*_MenuId*)

deletes menu

Arguments: deletemenu(+integer/symbol)

Succeeds: Deletes the entire menu *_MenuId*.

Fails: deletemenu fails if *_MenuId* is neither an integer identifying an existing menu nor a system menu symbol.

Note: Any of the menus, , **File**, **Find**, **Edit**, or **Window** can be deleted (the **Contexts menu** can also be deleted. Its ID is 32120). The menu ID is obtained when the menu is added using the predicate `addmenu`.

lastmenudata(_MenuId, _ItemId)

returns the menu ID and item ID of the last menu selection

Format: lastmenudata(_MenuId, _ItemId)

Arguments: lastmenudata(?integer, ?integer)

Succeeds: Unifies _MenuId and _ItemId with the menu and item IDs that were last selected with a `menuselect` event.

Fails: lastmenudata fails if either _MenuId and _ItemId are not variable, or do not unify with the ID integers of the last menu selection.

Note: The `userevent` predicate returns this data in symbol form. However, it is sometimes necessary to get the actual unambiguous selection integer values in order to resolve a selection, particularly if a menu contains multiple items with the same textual symbol.

menuitem(*_MenuId*, *_ItemId*, *_Item*, *_Attributes*)

sets or queries the item text and attributes

Arguments: menuitem(+integer/symbol, +integer, -symbol, ?attributes)

Succeeds: Sets or queries the name and attributes of the item *_ItemId* in the menu *_MenuId*. If one or both of the arguments *_Item* and *_Attributes* are variables, they will be instantiated to the corresponding menu item's value.

If *_Item* is a single hyphen (-) a dividing line is drawn across the full length of the menu. The dividing line may be disabled by using the appropriate metacharacters used to specify the attributes of an item. Refer to the predicate description of `additem` in this chapter for information on the metacharacters. However, the following variations are applicable to this predicate:

- Only nondefault value attributes will be reported in an attribute query.
- When setting an item's attributes, the attributes accumulate within a `menuitem` call and are applied in order.
- Unspecified attributes are not altered.
- Any existing icon, mark character or command key may be deleted by specifying the ASCII character null (zero, h00) as the character value for that attribute.
- Any character style specification replaces the existing style.
- An entire menu may be disabled or enabled by either specifying zero as both the item ID and as the attribute specification, or by specifying 0 (zero) as the attribute specification (no other attributes may be specified for menu titles). The *_Item* argument must be a variable in this case, since menu titles cannot be altered once defined.
- A hierarchical menu may be attached to any menu item by specifying the ASCII character ESC (27, h1B) as the character value for the command-key attribute of that item, and the ASCII character equivalent of the menu ID of the hierarchical menu as

the character value for the item-mark attribute of that item. (This limits hierarchical menu IDs to the range 0 to 255). The existence of a hierarchical menu and its menu ID may be determined by querying these attributes for the item in question.

Fails:

menuitem fails if

- `_MenuId` is neither an integer corresponding to a valid menu ID number nor a system menu symbol
- `_After_itemId` is neither the integer zero, the symbol `end_of_menu`, nor a valid item ID for an item in the menu specified
- `_Item` is not a variable
- `_Attributes` is neither a variable nor the symbol of attributes of the item `_ItemId`

Examples:

```
/* Add a tick mark to the 'Find Same' (item # 2) item in      */
/* the menu Find                                           */
*/

?- menuitem('Find', 2, _, '!√').
?- menuitem('Find', 2, 'Find Same', '!√').
YES
```

File	Edit	Find	Window	Contexts
		Find...	#	F
	√	Find Same	#	G
		Find Selection	#	H
		Replace...	#	R
		Replace Same	#	T
		Search Backwards	#	D

popupmenu(*_MenuId*, *_ItemId*, *_Top*, *_Left*)

displays a popup menu

Arguments: popupmenu(+integer, +integer, +integer, +integer)

Succeeds: Initiates and handles the display of a previously added popup menu having an ID *_MenuId*. The menu will be presented with the top-left corner of the item *_ItemId* located at the specified *_Top*, *_Left* absolute window coordinates when the mouse is pressed at that location.

Fails: popupmenu fails if

- *_MenuId* is not an integer corresponding to a valid menu ID number
- *_ItemId* is not an integer corresponding to the ID of an existing item in the menu *_MenuId*
- *_Top* and *_Left* are not integers specifying valid window coordinates
- the mouse is not pressed down when popupmenu is called

Note: This kind of menu can be used to respond to mouse clicks on user-defined buttons. If the button is in a graphics window, *_Top* is the y-axis and *_Left* is the x-axis.

Since mouse events are not detected by BNR Prolog in text windows, popupmenu will not work if the window coordinates are coordinates of a text window.

Examples:

```
?- addmenu(1, joe, -1),           % create a popupmenu
   addressitems('FONT', 1, end_of_menu),
   % add a list of fonts
   openwindow(graf, N, pos(_,_), size(_,_),
              options(zoomdocproc)),
   % open a graf window
   activewindow(N,graf), % make sure its active
   repeat,
       userevent(usermousedown, _, _X, _Y),
       % where's the mouse
       localglobal(N, _X, _Y, _Xa, _Ya),
       % translate to absolute coordinates
       popupmenu(1, 1, _Ya, _Xa),
       % pop up the menu
   fail.
   % exit with CTRL-<.>
```

Chapter 18

User Events

Effective Macintosh applications must be able to detect the occurrence of events created by the user (user events) from devices, such as the mouse or the keyboard, and then respond with some appropriate action. An event driven application in BNR Prolog does likewise: it polls the Macintosh for events and dispatches them to Prolog rules called event handlers. This chapter describes the predicates for polling user events. For further information on how to write event handlers refer to the chapter titled "User Interfaces" in the *BNR Prolog User Guide*.

Event Types

There are 13 distinct kinds of user events that fall into four categories:

Mouse events occur when the user presses (`usermousedown`) or releases (`usermouseup`) the mouse button in the content region of a graphics window. Other mouse events occur when the mouse is pressed in the drag, grow, zoom or close region of a (graphics or text) window (`userdrag`, `usergrow`, `userzoom` and `userclose`), or when a menu item is selected (`menuselect`).

Keyboard events occur when the user presses or releases a key (`userkey`).

Window events are generated when an active window is made inactive (`userdeactivate`), an inactive window is made active (`useractivate`) or when all or part of a graphics window is redrawn (`userupdate`).

Idle or null events are reported when there are no events pending (`userupidle` and `userdownidle`).

Event Reporting Priority

Some events have a higher reporting priority than others. Events generated by the system in order of reporting priority are

- userdeactivate
- useractivate
- menuselect, usermousedown, usermouseup, userdrag, usergrow, userzoom and userclose (all these have the same priority and are returned on a first come basis)
- userkey
- userupdate
- userupidle
- userdownidle

Predicates for Detecting User Events

There are two predicates for detecting user events:

lasteventdata	- returns information on the last user event
userevent	- detects a user event

```
lasteventdata(_Event, _Window, [_Mousegx, _Mousegy], _When,
[_Control, _Option, _Capslock, _Shift, _Command, _Mouseup])
```

returns information on the last user event

Arguments: `lasteventdata(?symbol, ?symbol, [?integer, ?integer], ?integer, [?integer, ?integer, ?integer, ?integer, ?integer, ?integer])`

Succeeds: Returns information on the event which terminated the last `userevent` predicate call. `_Event` is unified with the event. `_Mousegx`, `_Mousegy` are unified with the mouse coordinate pair, expressed in global coordinates. The `_When` argument is unified with the event's time stamp in processor clock ticks and the last list argument reflects the state of the modifier keys at the time the key is pressed. A "1" in the corresponding position indicates that the modifier key was pressed, otherwise the value is zero.

Fails: `lasteventdata` fails if any of the arguments do not unify with the data from the last event.

Examples:

```
/* Get information about the last event polled by the listener */
?- lasteventdata(_a, _b, _c, _d, _e).
   ?- lasteventdata(userkey, 'HDISK:Console', [181, 194], 678256,
[0, 0, 0, 0, 0, 1]).
YES
```

```
userevent(_Event, _Windowname, _Data1, _Data2)
userevent(_Event, _Windowname, _Data1, _Data2, noblock)
```

detects a user event

Arguments: userevent(?symbol, ?symbol, ?event_specific_type,
 ?event_specific_type)
 userevent(?symbol, ?symbol, ?event_specific_type,
 ?event_specific_type, +noblock)

Succeeds: The `userevent` predicates return user events when they occur. It can be invoked as a blocking or nonblocking call. If the `userevent` predicate without the `noblock` parameter is called, then Prolog execution is suspended until a user event occurs. Otherwise, if the symbol `noblock` is present as the fifth argument, an idle event (`userupidle` or `userdownidle` depending on the state of the mouse button) is returned if no user related event is pending and execution is continued. When an event occurs, all the information regarding the event is returned by means of the first four arguments: `_Event` is unified with the event which occurred; `_Windowname` is unified either with the window in which the event occurred or with the active window; `_Data1` and `_Data2` are event specific and their values depend on each of the events types described below.

`userdeactivate`

- `_Event` is unified with `userdeactivate` when the active window (the one in the foremost positions) is deactivated.
- The name of the window that was deactivated is unified with `_Windowname`.
- `_Data1` and `_Data2` are left unchanged.

`useractivate`

- `_Event` is unified with `useractivate` whenever a window is made active.
- The name of the window that was made active is unified with `_Windowname`.
- `_Data1` and `_Data2` are left unchanged.

menuselect

- `_Event` is unified with `menuselect` whenever a menu item is selected.
- `_Windowname` is unified with the active window.
- The menu name is unified with `_Data1` and the menu item is unified with `_Data2`.

usermouseup

- `_Event` is unified with `usermouseup` when the mouse button is released in an active window of type `graf`.
- `_Windowname` is unified with the name of the graphics window.
- `_Data1` and `_Data2` are unified with the x and y coordinate values (expressed in the local window coordinates) of the mouse respectively. Note that for a mouse-up event the coordinates may be less than zero or greater than the window's width or height, since the mouse button need not be released inside the window's content region.

usermousedown

- `_Event` is unified with `usermousedown` when the mouse button is pressed in the content region of an active window of type `graf`.
- `_Windowname` is unified with the name of the graphics window.
- `_Data1` and `_Data2` are unified with the x and y coordinate values (expressed in the local window coordinates) of the mouse respectively. These coordinate values will always be greater than zero and less than the window's width and height. If the window in which the mouse click occurs is not the active window then the window will be made active and the associated mouse-down and mouse-up events will be consumed internally (see `useractivate` and `userdeactivate` above).

userdrag, usergrow, userzoom, userclose

- `_Event` is unified with one of the above values when a mouse down event is detected in the drag, grow, zoom or close regions of the active window.
- `_Windowname` becomes unified with the name of the active window.
- `_Data1` and `_Data2` are unified with the x and y coordinates (expressed in global coordinates) of the mouse-down position of the event.

`userkey`

- `_Event` is unified with `userkey` when a keyboard key is pressed.
- `_Windowname` is unified with the name of the active window.
- `_Data1` is unified with the single character symbol.
- `_Data2` is unified with a list of six elements defining the state of the modifier keys at the time the key is pressed. It has the form `[_Control, _Option, _Capslock, _Shift, _Command, _Mouseup]`. A "1" in the corresponding position indicates the modifier key was pressed, otherwise, the value is "0" (zero). Key events are generated for all keyboard keys except the modifier keys (control, shift, caps locks, option and command) which internally alter the key code values of the keys before they are passed on.

`userupdate`

- `_Event` is unified with `userupdate` when a previously covered portion of a graphics window is uncovered for any reason. This includes the time when a graph window is first opened and displayed, since, by definition, the window is uncovered for the first time and therefore needs to have its contents updated. The normal response to a `userupdate` event is to redraw the contents of the window (using `dograf`); the system automatically restricts the effects of redrawing to the update region of the affected window until the next call to `userevent`, at which time the entire window is updated.
- `_Windowname` is unified with the name of the graphics window.
- `_Data1` and `_Data2` are left unchanged.

`userdownidle, userupidle`

- `_Event` is unified with either `userdownidle` or `userupidle` when the `noblock` symbol is specified and no other user event has occurred.
- `_Event` will unify with `usedownidle` if the mouse button is currently pressed down, otherwise it will unify with `userupidle`.
- `_Windowname` is unified with the name of the active window.
- `_Data1` and `_Data2` are unified with the x and y coordinate values (expressed in the local window coordinate system) of the mouse respectively. The coordinates may be less than zero or greater than the window's width or height since the mouse need not be inside the window's content region.

- Fails:** `userevent` fails if
- `_Event` is neither a variable nor a symbol representing one of the event types
 - `_Windowname` is neither a variable, nor a symbol which is the name of the the active window or the name of window in which the event occurred
 - `_Data1` and `_Data2` are neither variables nor valid event specific values

Note 1: `usermouseup`, `usermousedown` and `userupdate` are reported only if they occur in a window of type `graf`.

Note 2: `userdownidle` or `userupidle` is returned any time a nonblocking call is made and no other user event is pending.

Note 3: A `userupdate` event can occur at any time.

Examples:

```

/* The following example shows that opening a graphics      */
/* window causes 3 events to occur. The fourth call       */
/* to userevent is satisfied by clicking in the window     */
?-
[openwindow(graf, test, pos(_, _), size(_, _), options()),
  userevent(_a, _b, _c, _d),
  userevent(_e, _f, _g, _h),
  userevent(_i, _j, _k, _l),
  userevent(_m, _n, _o, _p)].
?- [openwindow(graf, test, pos(40, 80), size(566, 321), options()),
  userevent(userdeactivate, 'HDISK:Console', _c, _d),
  userevent(useractivate, test, _g, _h),
  userevent(userupdate, test, _k, _l),
  userevent(usermousedown, test, 146, 138)].
YES

/* this example polls for user events until the mouse is   */
/* clicked on the window drag region and beeps otherwise.  */
:- [repeat, ( [userevent(userdrag, _, _, _), !]
; [beep, fail])].

```

Chapter 19

Predefined Dialogs

Dialog boxes are used by an application to convey important information to the user, or to request information needed to complete a command from the user.

Dialogs can be either modal or modeless.

A *modal* dialog restricts the user's action. The user must respond to the dialog before proceeding with the application. Clicking the mouse outside the dialog causes the system to beep. The user is usually provided with the option of cancelling the dialog box, typing some text from the keyboard or manipulating controls within the dialog box with the mouse. All of the predefined dialogs operate in this mode.

A *modeless* dialog does not require the user's immediate response and behaves in the same way as any other document window. The user can open another window and work with it, or execute another command. The process that created the modeless dialog is suspended until the user reactivates it by choosing its **OK** or **CANCEL** button.

Predicates for Creating Dialogs

<code>confirm</code>	- displays a message requiring confirmation
<code>message</code>	- displays a message
<code>namefile</code>	- displays a file specification dialog
<code>query</code>	- displays a dialog prompting a read
<code>select</code>	- displays a list of items for multiple selection
<code>selectfile</code>	- displays a file selection dialog
<code>selectone</code>	- displays a list of items for single selection

confirm(_Prompt, _Cancel_enabled, _Default_response, _Response)
displays a message requiring confirmation

Arguments: confirm(+symbol, +symbol, +symbol, ?symbol)

Succeeds: Displays the text `_Prompt` in a dialog box containing **OK** and **NO** buttons, and optionally a **CANCEL** button. The **CANCEL** button is displayed if the value of the argument `_Cancel_enabled` is the symbol 'YES'.

The button designated by `_Default_response` is outlined in bold and is the response entered if either the *enter* or *return* key is pressed. The values allowed for `_Default_response` are the symbols 'YES', 'NO' and 'CANCEL'. The default response button may be user selected by either clicking the mouse on them, or by typing in the first character of the desired response.

The values allowed for `_Response` are the symbols 'YES' and 'NO'.

`confirm` succeeds if the user selects either the **YES** or **NO** button and `_Response` is unified with the selected value.

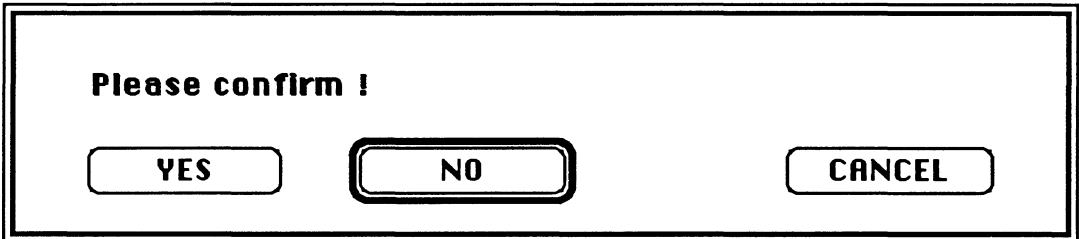
Fails:

`confirm` fails if

- `_Prompt` is not a symbol
- `_Cancel_enabled` is neither the symbol 'YES' nor the symbol 'NO'
- `_Default_response` is not one of the allowed symbols ('YES', 'NO' and 'CANCEL')
- `_Response` is neither variable nor the symbol 'YES' or 'NO'
- the user selects the **CANCEL** button

Examples:

```
/* This call */  
:- confirm('Please confirm !', 'YES', 'NO', _Response).  
YES  
/* will display the following dialog box */
```



message (_Text)

displays a message

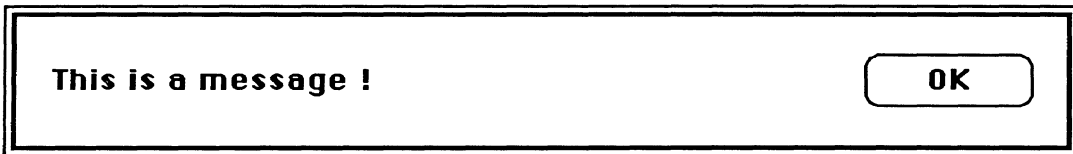
Arguments: message (+symbol)

Succeeds: A modal dialog is displayed. The dialog box contains the text _Text and an **OK** button. The user must click on the **OK** button or press either the *enter* or *return* key to continue.

Fails: message fails if _Text is not a symbol.

Examples:

```
/* This call                                     */
:- message('This is a message').
YES
/* will display the following dialog box        */
```



nameafile(_Doit_label, _Filefield_label, _Default_response, _Response)

displays a file specification dialog

Arguments: nameafile(+symbol, +symbol, +filename, ?fullfilename)

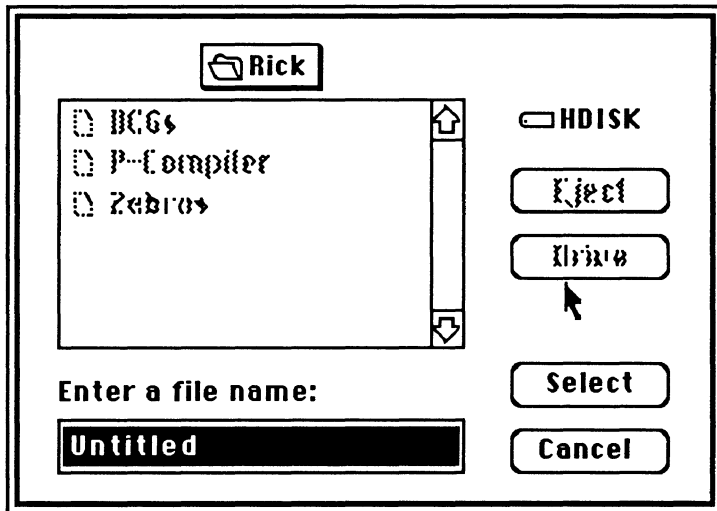
Succeeds: Displays a file specification dialog containing a directory browser and an editable filename field. Users may browse through the disk file directories and eventually either specify a file or select **CANCEL**. The **Doit** button in the dialog is given the label `_Doit_label` (for example, 'Save' or 'Delete') and the filename field is given the label `_Filefield_label` and primed with the default response `_Default_response`. The default response should be a filename, not a full filename. Any leading directory name components in the default response will be stripped out before being displayed. The response output `_Response` is the full filename of the user specified file. This file may or may not exist. If it does, an explicit subdialog will be used to confirm the selection.

Fails: nameafile fails if

- `_Doit_label` is not a symbol
- `_Filefield_label` is not a symbol
- `_Default_response` is not a valid Macintosh file name
- **CANCEL** is selected

Examples:

```
/* This call */  
:- namefile('Select', 'Enter a file name:',  
'Untitled', _Response).  
YES  
/* displays the following dialog box */
```



```
query(_Prompt, _Default_response, _Response)
```

display a dialog box prompting a read

Arguments: query(+symbol, +symbol, ?symbol)

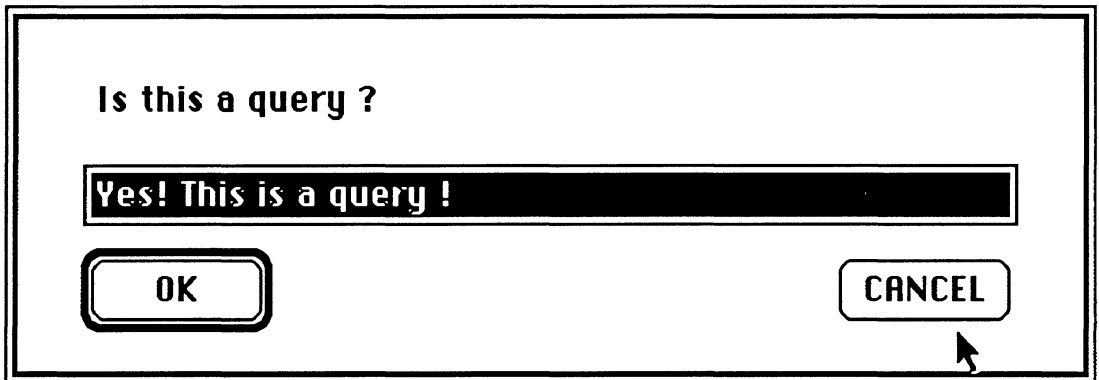
Succeeds: Displays a modal dialog box with a text box containing a prompt message `_Prompt`, an editable text box containing a default response `_Default_response`, an **OK** button and a **CANCEL** button. The default response may then be edited to produce the desired response `_Response`. `query` succeeds when the user clicks on the **OK** button, or presses either the *enter* or *return* key.

Fails: `query` fails if

- `_Prompt` is not a symbol
- `_Default_response` is not a symbol
- the user selects the **CANCEL** button

Examples:

```
/* This call                                                    */
:- query('Is this a query?', 'YES, it is!', _Response).
YES
/* displays the following dialog box                            */
```



```
select (_Title, _Choicelist, _Selection)
select (_Title, _Choicelist, _Selection, _Initialselection)
```

displays a list of items for multiple selection

Arguments: select(+symbol, +list , ?list)
 select(+symbol, +list, ?list, +list)

Succeeds: Displays a dialog box containing the list of the items in
 _Choicelist, and an **OK** and **CANCEL** button. _Selection is
 unified with the list of items selected by the user from
 _Choicelist.

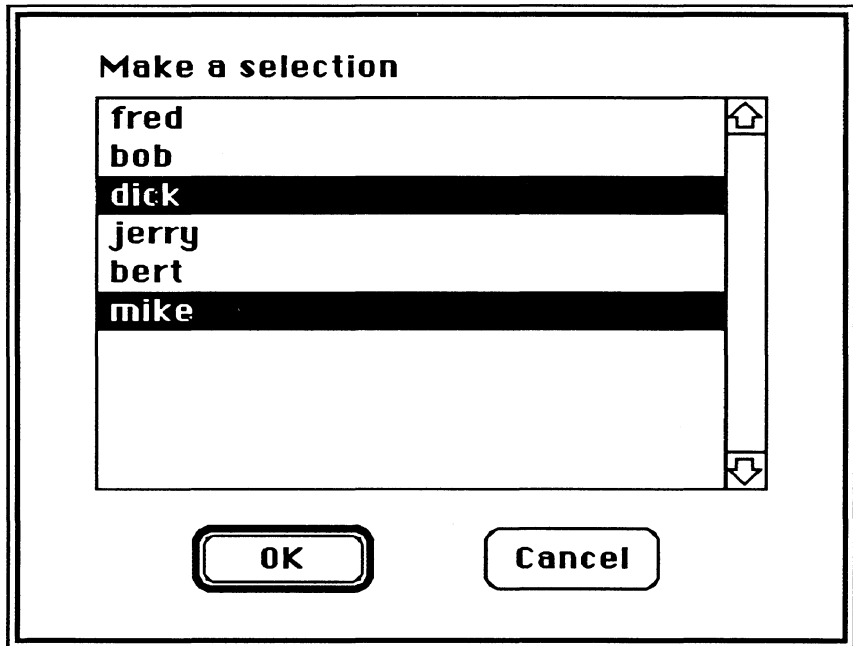
If an initial selection list _Initialselection is specified, then these items are initially highlighted, otherwise the first item is highlighted. A single item may be highlighted by clicking on the item. Multiple items are highlighted by pressing the *command* key (also called the Apple key) while clicking on the items. A block of items can be selected by either clicking on the first item and then pressing the *shift* key while clicking on the last item, or by dragging the mouse across the items while holding the mouse button down. The highlighted items may be selected by clicking on the **OK** button, or by pressing *enter* or clicking on it. A single item may also be selected directly by double clicking on it.

Fails: select fails if

- _Title is not a symbol
- the elements of the list _Choicelist are not symbols
- _Initialselection is not a sublist of the list _Choicelist

Examples:

```
/* This call */  
  
:- select('Make a selection', [fred, bob, dick, jerry,  
bert, mike], _Selection, [dick, mike]).  
YES  
/* displays the following dialog box */
```



See Also: `selectone` in this chapter.

selectfile(_Filetype, _Doit_label, _Selection)

displays a file selection dialog

Arguments: selectfile(+filetype/list_of_filetypes, +symbol, ?fullfilename)

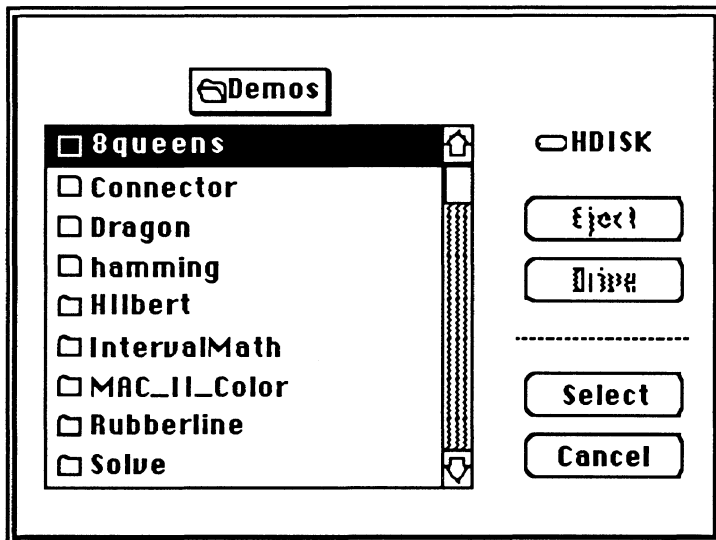
Succeeds: Displays a file selection dialog containing a list of files of the designated file types (for example, 'TEXT', 'APPL'). Users may browse through the disk file directories and eventually select a file or select **CANCEL**. The argument `_Filetype` may be either a single item or a list of items. Each item should be a four-character symbol specifying a Macintosh file type. A single null symbol or an empty list is interpreted as any or all file types. The **Doit** button is given the label specified by the argument `_Doit_label`. `_Selection` is unified with the full filename of the selected file.

Fails: selectfile fails if

- `_Filetype` is neither a null symbol, a Macintosh file type or a list of zero or more file types
- `_Doitlabel` is not a symbol
- the user selects **CANCEL**

Examples:

```
/* This call */
:- selectafile(['APPL', 'APWS'], 'Select', _Selection).
YES
/* displays the following dialog box */
```



selectone(_Title, _Choicelist, _Selection)
selectone(_Title, _Choicelist, _Selection, _Initialselection)

displays a list of items for single selection

Arguments: selectone(+symbol, +list, ?symbol,)
 selectone(+symbol, +list, ?symbol, +list)

Succeeds: Displays a dialog box containing the list of the items in
 _**Choicelist**, an **OK** button and a **CANCEL** button. **_Selection** is
 instantiated to the item selected by the user from **_Choicelist**.

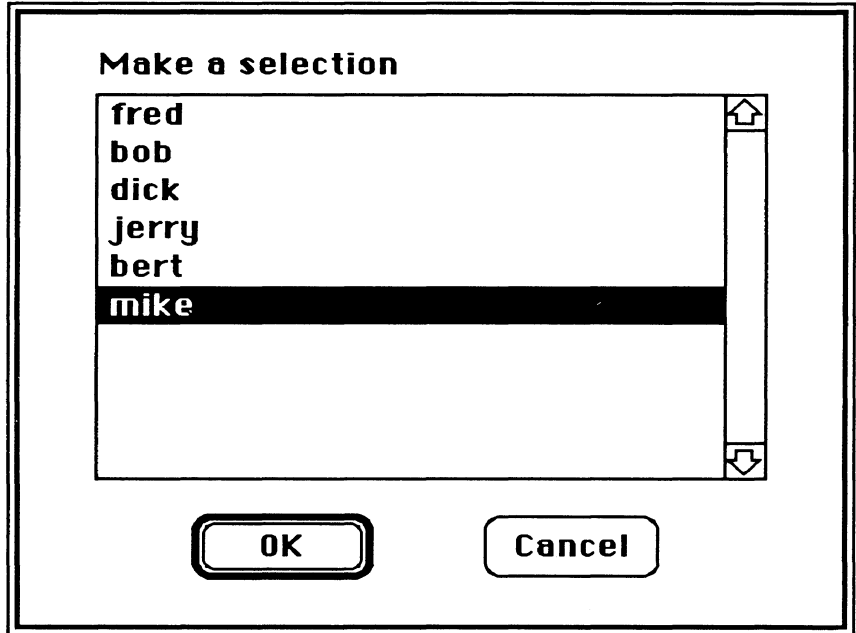
 If an initial selection **_Initialselection** is specified then this
 item is initially highlighted, otherwise the first item is highlighted.
 An item may be highlighted by clicking on the item. The
 highlighted item is then selected by clicking on the **OK** button, or by
 either pressing the *enter* key or clicking on the item. A single item
 may also be selected directly by double clicking on it.

Fails: selectone fails if

- **_Title** is not a symbol
- the elements of the list **_Choicelist** are not symbols
- **_Initialselection** is not a sublist of the list **_Choicelist**

Examples:

```
/* This call */  
:- selectone('Make a selection' [fred, bob, dick,  
jerry, bert, mike], _Selection, mike).  
YES  
/* displays the following dialog box */
```



See Also: `selectone` this chapter.

Chapter 20

Macintosh System Utility Predicates

This chapter documents predicates which provide access to miscellaneous Macintosh system facilities.

System Utility Predicates

beep	- causes an audible beep
deskaccessory	- invokes a desk accessory
doubletime	- queries or sets the mouse double-click time
getappfiles	- queries the list of files selected at application launch
invalidrect	- invalidates a rectangular region in a window
isfont	- defines a font
listfonts	- queries the list of installed font numbers
localglobal	- translates between global and local window coordinates
mbarheight	- queries the height of the menu bar
messagebutton	- sets or queries the label on the activity button
scrapcontents	- queries the type of data in the scrap
scrndimensions	- queries the dimensions of the Macintosh monitor screen
setcursor	- sets the cursor ID
validrect	- validates the rectangular region in a window

Descriptions of the predicates follow.

beep

causes an audible beep

Arguments: None.


Succeeds: beep causes the system to beep. The volume of the beep depends on the current setting of the speaker which is adjusted by means of the Control Panel desk accessory. If the volume of the speaker is set to zero, no sound is heard and the menu bar flashes instead.

Fails: Never fails.

deskaccessory (*_Name*)

invokes a desk accessory

Arguments: deskaccessory(+symbol)

Succeeds: Invokes the desk accessory *_Name*. (The Macintosh operating system gives no notification that the desk accessory was successfully run.) The names of desk accessories are normally acquired from the  menu selections. By convention, these names have a null character (`\00,`) as the first character of their name.

Fails: deskaccessory fails if *_Name* is not a symbol.

Examples:

```
?- deskaccessory('\00Calculator').  
?- deskaccessory('Calculator').  
YES
```

doubletime(_Deltatime)

queries the mouse double-click time

Arguments: doubletime(?integer)

Succeeds: Unifies _Deltatime with the mouse double-click time which has been set by the Macintosh control panel.

Fails: doubletime fails if _Deltatime does not unify with the mouse double-click time.

Examples:

```
?- doubletime(_).  
    ?- doubletime(32).  
YES
```

```
?- doubletime(32).  
    ?- doubletime(32).  
YES
```

getappfiles (*_Filelist*)

queries the list of files selected at application launch

Arguments: getappfiles(?list)

Succeeds: Unifies *_Filelist* with a list of the full filenames of any files selected in the finder when the application was launched. getappfiles is used mainly by applications written in Prolog.

Fails: getappfiles fails if *_Filelist* does not unify with the list of start-up file names.

Examples:

```
?- getappfiles(_).
   ?- getappfiles([appfile1, appfile2]).
YES

?- getappfiles([appfile1, appfile2]).
   ?- getappfiles([appfile1, appfile2]).
YES

?- getappfiles([appfile2, appfile1]).
NO
```

invalidrect(_Windowname, _Left, _Top, _Right, _Bottom)

invalidates a rectangular region in a window

Arguments: invalidrect(+symbol, +integer, +integer, +integer, +integer)

Succeeds: Invalidates the rectangular region specified by _Left, _Top, _Right and _Bottom in the window _Windowname. This region is accumulated into the current update region for that window and will cause an update event to be generated for the window.

Fails: invalidrect fails if

- _Windowname is not the name of a window
- _Left, _Top, _Right and _Bottom are not integers

isfont(*_Number*, *_Point_size*, *_Name*, *_Size_detail*)***defines a font***

Arguments: isfont(?integer, ?integer, ?symbol, ?list)**Succeeds:** Defines a relationship between a font number *_Number*, a point size *_Point_size*, a font name *_Name* and the size detail *_Size_detail* at that point size. *_Size_detail* is a list of the form [*_Ascent*, *_Descent*, *_Leading*, *_Max_width*]. *_Point_size* can be of any value and the Macintosh toolbox will scale the nearest available size of a "real" font of that name to the requested size to get the size detail.**Fails:** isfont fails if

- *_Number* is neither a variable nor an integer
- *_Point_size* is neither a variable nor an integer
- *_Name* is neither a variable nor a symbol
- *_Size_detail* is neither a variable nor a list
- no such relationship exists between the arguments specified

Examples:

```
/* generate all font information                                     */
?- isfont(_..).
  ?- isfont(2, 9, 'New York', [10, 2, 0, 10]).
  ?- isfont(2, 10, 'New York', [10, 2, 0, 10]).
  ?- isfont(2, 12, 'New York', [12, 3, 1, 13]).
      .
      .
YES
```

```
/* generate all font number 2 information */
?- isfont(2, _..).
   ?- isfont(2, 9, 'New York', [10, 2, 0, 10]).
   ?- isfont(2, 10, 'New York', [10, 2, 0, 10]).
      .
      .
YES

/* generate all fonts with a point size of 18 */
?- isfont(_, 18, _..).
   ?- isfont(2, 18, 'New York', [17, 14, 2, 20]).
   ?- isfont(3, 18, 'Geneva', [18, 4, 1, 18]).
   ?- isfont(20, 18, 'Times', [14, 4, 1, 18]).
      .
      .
YES

/* generate information about New York fonts */
?- isfont(_, _, 'New York', _).
   ?- isfont(2, 9, 'New York', [10, 2, 0, 10]).
   ?- isfont(2, 10, 'New York', [10, 2, 0, 10]).
      .
      .
YES

/* generate information about a scaled font */
?- isfont(2, 100, _..).
   ?- isfont(2, 100, 'New York', [88, 21, 13, 108]).
YES
```

See also: listfonts in this chapter.

listfonts (_Fontlist)

queries the list of installed fonts numbers

Arguments: listfonts(?list)

Succeeds: Unifies _Fontlist with the list of installed font numbers.

Fails: listfonts fails if _Fontlist does not unify with the list of installed font numbers.

Examples:

```
/* generate all font information                                     */
?- listfonts(_).
   ?- listfonts([2, 5, 0, 3, 4, 20, 22]).
YES

?- listfonts([2, 5, 0, 3, 4, 20, 22]).
   ?- listfonts([2, 5, 0, 3, 4, 20, 22]).
YES

?- listfonts([2, 0, 5, 3, 4, 20, 22]).
NO
```

See Also: isfont in this chapter.

localglobal(_Windowname, _Xlocal, _Ylocal, _Xglobal, _Yglobal)

translates between global and local window coordinates

Arguments: localglobal(+symbol, ?number, ?number, ?number, ?number)

Succeeds: Translates between the local coordinates of the named window and the global coordinates of the Macintosh screen.

Fails: localglobal fails if

- _Windowname is not the name of a window
- both _Xlocal and _Xglobal are variables
- both _Ylocal and _Yglobal are variables
- _Xlocal, _Ylocal, _Xglobal, _Yglobal are neither variables nor numbers
- the local coordinates specified do not correspond to the global coordinates specified

Examples:

```
?- localglobal(fred, 30, 40, _, _).
   ?- localglobal(fred, 30, 40, 60, 194).
YES

?- localglobal(fred, _, _, 60, 194).
   ?- localglobal(fred, 30, 40, 60, 194).
YES

?- localglobal(fred, 30, _, _, 194).
   ?- localglobal(fred, 30, 40, 60, 194).
YES

?- localglobal(fred, 30, _, 60, _).
NO
```

mbarheight (_Height)

queries the height of the menu bar

Arguments: mbarheight(?integer)

Succeeds: Unifies _Height with the height of the menu bar in pixels.

Fails: mbarheight fails if _Height is neither an integer equal to the height of the menu bar, nor a variable.

`messagebutton(_Label, _Key)`

sets or queries the label on the activity button

Arguments: `messagebutton(?symbol, ?symbol)`

Succeeds: If `_Label` or `_Key` is a variable then `messagebutton` unifies with the current system activity button label or key. If `_Label` or `_Key` is a single character symbol, then `messagebutton` sets the label or key.

The system activity label is the string displayed activity field in the bottom-left corner of the active text window. When the mouse is clicked on the activity field, an event of type `userkey` is generated and the value of the key is set by the `_Key` argument to `messagebutton`.

Fails: `messagebutton` fails if

- `_Label` is neither a variable nor a symbol
- `_Key` is neither a variable nor a single character symbol

Examples:

```
/* query current status */
?- messagebutton(_L, _K).
   ?- messagebutton(running, '').
YES

/* set message to up and key to up cursor */
?- messagebutton(up, '\1E').
   ?- messagebutton(up, ' ').
YES

?- messagebutton(_, ab).
NO
```

See Also: `userevent` in the chapter titled "User Events".

scrapcontents (*_Type*)

queries the type of data in the scrap

Arguments: `scrapcontents(?symbol)`

Succeeds: Unifies *_Type* with the type of data in the scrap: 'TEXT' or 'PICT'.
If the scrap contains both types of data, *_Type* is unified with 'PICT'.

Fails: `scrapcontents` fails if

- *_Type* is neither a variable nor one of the symbols 'TEXT' or 'PICT'
- there is no data in the scrap

`scrndimensions(_Width, _Height)`

generates the dimensions of the Macintosh monitor

Arguments: `scrndimensions(?integer, ?integer)`

Succeeds: Unifies `_width` and `_Height` with the dimensions (in pixels) of the Macintosh monitor.

Fails: `scrndimensions` fails if `_width` and `_Height` are neither variables nor the dimensions of the monitor.

Examples:

```
/* query current status                                     */
?- scrndimensions(_W, _H).
   ?- scrndimensions(512, 342).
YES

/* set message to up and key to up cursor                 */
?- scrndimensions(512, 342).
   ?- scrndimensions(512, 342).
YES
```

setcursor (*_Id*)

sets the cursor ID

Arguments: setcursor(+integer/symbol)

Succeeds: If *_Id* is an integer (interpreted as the resource ID of a cursor resource) or one of the predefined symbols, *cross*, *plus*, *ibeam*, *watch* or *arrow*, then the cursor ID will be set to that value. The cursor displayed will be changed accordingly.

Fails: setcursor fails if *_Id* is not one of the predefined symbols, or the ID of a cursor resource.

validirect (*_Windowname*, *_Left*, *_Top*, *_Right*, *_Bottom*)

validates the rectangular region in a window

Arguments: validirect(+symbol, +integer, +integer, +integer, +integer)

Succeeds: validirect validates the rectangular region specified by *_Left*, *_Top*, *_Right* and *_Bottom* in the window *_Windowname*. This removes the region from the current update region for that window if there was any intersection of the two regions.

Fails: validirect fails if

- *_Windowname* is not the name of a window
- any of the coordinates specified by *_Left*, *_Top*, *_Right* and *_Bottom* are not integers.

See Also: invalidirect in this chapter.

Chapter 21

External Language Interface

Primitives can be written in other languages (for example, Pascal and C), and called directly from BNR Prolog. To facilitate the interface between languages, the primitives must be compiled as code resources, and then placed in the resource fork of a file with the resource type 'PEXT' (Prolog EXTernal). *Macintosh Programmer's Workshop* and the *Lightspeed* family of language products are examples of development environments which provide tools for generating code resources.

Pascal and C examples which use the external language interface are provided in the Chapter titled "Foreign Language Interface" in the *BNR Prolog User Guide*.

Calls to Foreign Languages

Accessing External Primitives

External primitives are defined to the Prolog system by using the `defexternal` predicate. This predicate loads the code resource into memory and creates a Prolog clause to call it. The code resource will remain in memory until the context containing the associated Prolog clause is removed.

defexternal(_Clausehead, _Filename, _Resourcename, _In, _Out)

allows a code resource to be called as a Prolog goal

Arguments: defexternal(+clausehead, +filename, +symbol, +list, +list)

Succeeds: Loads the resource `_Resourcename` of type 'PEXT' from the resource fork of the file `_Filename`, and binds `_Clausehead` to the external definition. `_Clausehead` is of the form `_Procname(_Variables..)` where `_Procname` is the name by which Prolog clauses can call the external and `_Variables..` are the arguments to the external. If `_Filename` is the null symbol (`''`), then the current applications file is searched for the specified resource. `_In` and `_Out` are lists defining the input and output arguments and their types. Items in these lists are of the form `_Name : _Type`, where `_Name` is the name of an argument in `_Variables` and `_Type` is integer, float, symbol or bucket.

Fails: defexternal fails if

- `_Clausehead` is not a valid clause head
- `_Resourcename` does not reside in the file `_Filename`
- the elements of `_In` and `_Out` are not of the correct form

Examples:

```
/* loads a code resource "freemem" from the application file */
/* and creates a clause name "free_memory" which is used to */
/* call it. The routine returns two integers. */

defexternal(free_memory(_LargestFree, _TotalFree),
            '', 'freemem', [],
            [_TotalFree :integer, _LargestFree : integer]).

/* An example of a call to free memory */
?- free_memory(_Largest, _Total).
?- free_memory(102488, 148796).
YES
```

Data Types

Currently the following types of data are available to external routines:

- integer
- float
- symbol
- bucket

An integer is passed as a 32-bit value that is of type `longint` in Pascal, or type `long` in C. The value of the integer must be representable in 29 bits, including the sign, which is in the range of -26843556 to 268435455.

A float is passed as a 32-bit pointer to an 80-bit extended Standard Apple Numerics Environment (SANE) format number that is represented as an `^extended` in Pascal, and a `*extended` in C. The value of the number is maintained internally as an 8-bit exponent with a 20-bit mantissa, so some accuracy may be lost in conversions.

A symbol is passed as a 32-bit pointer to a Pascal string which contains a length byte followed by up to 255 characters in the string, with each character using a byte. In Pascal, the declaration is `^string[255]`.

A bucket is passed as a 32-bit value that is of type `longint` in Pascal, and type `long` in C. Buckets are not the same as integers and do not unify with them.

Parameter Interface

Each external definition is passed a pointer to a structure that contains the result, a user definable handle, the input parameters, and the output parameters. The input and output parameters are passed in the order specified inside the input and output lists and not by the order inside the clause-head. The structure for `free_memory` described in the example would be as follows (Pascal interface):

```
StackFrame = RECORD
  Result      : LONGINT;
  Reserved    : LONGINT;
  UserHandle  : HANDLE;
  TotalFree   : LONGINT;
  LargestFree : LONGINT;
END;
```

where

`Result` is initially zero which indicates failure. Nonzero indicates success of the procedure call.

`Reserved` is a field saved for future use.

The value of the `UserHandle` is initially NIL, but may be updated by the external, and is maintained by the system between calls to the external. It is passed to the procedure with every call and can be used by a procedure to maintain global data structures. When the context containing the external procedure call clause is exited, the space pointed to by `UserHandle` is freed. `UserHandle` should only be used as a handle.

In the case of both input and output parameters, the space for all strings and floating point numbers is allocated by BNR Prolog before calling the routine.

In either Pascal or C, the structure appears as the lone parameter to the external procedure, as follows:

```
PROCEDURE Proc (VAR P : StackFrame);
```

or

```
void Proc (P)  
struct StackFrame *P;
```

Restrictions

External procedures that are accessed from BNR Prolog cannot have global data. Any data that is intended as global should be defined and manipulated through the `UserHandle` in the `StackFrame`. `UserHandle` must be maintained as a handle, if it is to be used at all.

External procedures need not be concerned with popping parameters off the stack on return. BNR Prolog will restore the stack if necessary.

External routines must be procedures rather than functions. There must be no more than one parameter to an external procedure, the pointer to the parameter structure. If the external procedure is defined as a function or more parameters are used, unexpected results may occur.

Pointer parameters should not be modified. Such action will result in unrecoverable space in memory.

Appendix A Macintosh Extended Character Set

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	sp	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~	DEL	Ä	Å
130	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã
140	â	ç	é	è	ê	ë	í	ì	î	ï
150	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
160	†	°	¢	£	§	•	¶	ß	®	©
170	™	'	"	≠	Æ	Ø	∞	±	≤	≥
180	¥	μ	∂	Σ	Π	π	∫	ª	º	Ω
190	æ	ø	¿	¡	¬	√	f	≈	Δ	«
200	»	...		À	Á	Ö	Œ	œ	-	—
210	"	"	'	'	+	◊	ÿ	ÿ	/	□
220	◁	▷	fi	fl	‡	·	,	„	‰	Å
230	Ê	Á	Ë	È	Í	Ì	Î	Ĭ	Ó	Ô
240	⌘	Ò	Ú	Û	Ü	ı	ˆ	˜	-	˘
250	·	·	·	·	·	·	·	·	·	·

Appendix B

Error Messages

Run Errors

Below is a list of run time errors. These errors are output from BNR Prolog when an execution error occurs. The errors are ordered by error number for easy searching.

- 1: Global stack dangling reference
- 2: World stack dangling reference
- 3: Bad record on global stack
- 4: Bad record on world stack
- 5: Global stack full
- 6: World stack full
- 7: Control stack full
- 8: Bad record
- 9: Structure too deep
- 10: Invalid tag
- 11: Division by zero
- 13: User aborted execution
- 14: Bad primitive
- 15: Buffer length exceeded
- 16: Non executable term
- 17: Integer overflow
- 18: Bad clause
- 19: Not yet implemented
- 20: Looped list

392 Appendix B: Error Messages

- 22: Odd number of parser classes
- 24: Too many variables in clause
- 41: State space full
- 42: Cannot extend state space
- 43: State space double limit
- 44: Cannot write structure
- 47: Operator not yet defined for INFIX call
- 48: Unable to make infix operator
- 49: Unable to find strings for comparison
- 50: Unable to find string for hashing
- 51: File does not contain a valid state space
- 52: Bad item in loaded state space
- 53: Total size of state space inconsistent
- 54: Attempt to allocate cell too small
- 55: Free list in state space corrupted
- 57: Attempt to unreference a non-string
- 58: Attempt to reference a non-string
- 59: Attempt to dissolve variable list
- 60: Attempt to dissolve unrecognized term
- 61: Could not find string
- 62: Could not lookup string
- 63: Attempt to copy variadic structure
- 64: Unable to find string to compare
- 65: Unable to forget item
- 67: Unable to copy intervals
- 68: Real number overflow
- 69: State Space : Unable to release memory
- 70: State Space : Unable to obtain memory

Syntax Errors

The following is a list of syntax errors output by BNR Prolog.

- 2: Waiting for rest of term.
- 3: Expecting atom, variable or anonymous as functor name
- 4: Operand stack underflow
- 5: Operator stack underflow
- 6: Too many variables in clause - limit is 255
- 7: Parser bug. Please report.
- 8: Integer overflow
- 9: Integer expected to follow unary minus
- 10: Digit expected
- 11: Maximum nesting depth exceeded for lists and structures - limit is 40
- 12: Too many right parentheses ')' for expression
- 13: Need an operand before this last symbol
- 14: Empty pair of parentheses '()'
- 15: Illegal use of parentheses '()'
- 16: Bad character in this symbol
- 17: Not a tail variable
- 18: Clause being parsed is too big
- 19: No operand between pair of operators
- 20: This symbol not an operator, so a comma needed before it
- 21: Comma needed before left parenthesis '('
- 22: Comma needed before current list or structure
- 23: Comma or operator needed before left parenthesis '('
- 24: Mismatch of bracket types

394 Appendix B: Error Messages

- 25: This symbol is not a real number
- 26: String length cannot exceeded 255 characters
- 27: No predicate after comma
- 28: Need comma, prefix or infix operator, (or [before this symbol
- 29: Need operand,),], or postfix operator before this symbol
- 30: This operator and previous one have incompatible types or precedences
- 31: This variable name is reserved
- 32: Token is too long
- 33: Character in current token cannot be interpreted
- 34: Incomplete expression
- 35: Abandoned parsing of this structure
- 36: Unrecognized parsing action
- 37: Ignoring extra right brackets/parentheses at end of clause
- 38: Adding matching right brackets/parentheses at end of clause
- 39: Need closing right bracket for |-list
- 40: Use right bracket] to end lists
- 41: Use right parenthesis) to end structures

System Errors

A complete list of the Macintosh file system error codes is presented in Appendix A of Volume III of *Inside Macintosh*. Examples of this type of error include: I/O error, too many files open, bad filename, file is locked, and disk is full. In addition to the Macintosh file system error codes, a number of additional codes have been defined by the Prolog system. These are listed below:

MaxDocErr	=	-200; {Maximum # of documents exceeded}
UserWindErr	=	-201; {Illegal operation on a user defined window}
UnkEvErr	=	-202; {Unknown or unexpected event type seen}
WinOflwErr	=	-203; {Implementation restriction, Windows <= 32k}
UnImplErr	=	-204; {Unimplemented or inaccessible routine}
IntMMIerr	=	-205; {Internal MMI error}
ConsOpErr	=	-206; {Illegal operation on the Console window}
ProBusyErr	=	-207; {Open prolog stream can't be closed}
UserCanErr	=	-208; {User 'Cancel'.}

Appendix C

Compatibility with other Prologs

Although BNR Prolog is largely compatible with other Prologs in the Edinburgh family, there are some syntactic and semantic differences. The following is a brief description of the differences that may affect the execution of programs. Please refer to the chapter titled "Prolog Compatibility Issues" in the *BNR Prolog User Guide* for further details.

Clause Bodies

Clause bodies in BNR Prolog are lists, rather than comma-structures. Thus the clause

```
p :- a,b,c.
```

in BNR Prolog unifies with the term

```
:-(p, [a,b,c]).
```

not with

```
:-(p, ('', '(a, ', '(b,c))))).
```

Lists

The variable remainder of a list in BNR Prolog is a tail variable. The term `[A|B]` unifies with `[A,B..]` and `B` unifies with the list `[B..]`, which contains a tail variable. Thus the questions

```
?- [a|Tail], Tail=x .
```

```
?- [a|Tail], var(Tail).
```

will fail because `Tail` must be a list.

Nonempty lists in BNR Prolog do not unify with the structure `(Head . Tail)` as they do in some other Prologs. The period `(.)` is not an operator.

Operators

Operators in BNR Prolog are declared by adding `op/3` facts in the clause space, rather than executing them as goals or directives.

An operator that is declared infix must have the same precedence number as its prefix or postfix form. The same operator cannot be both prefix and postfix.

To *mention* an operator in a program, it may need to be quoted. Since it must be possible to *use* operators without the quote marks, symbols that must always be quoted (for example, 'a b c') cannot be declared as operators.

The operator `"=="` in BNR Prolog is synonymous with the arithmetic comparison operator `"=:"`. All programs that expect `"=="` to perform a term comparison should use the operator `@=` instead. The operators `"\=="` and `"@\"=` are synonymous, as are `"=\=` and `"<>"`.

Comma `,` is treated as a separator unless it is specifically declared as an operator and used with explicit parentheses.

Strings

There are no strings in BNR Prolog: the term `"Hello"` does not unify with the list of characters `[72, 101, 108, 108, 111]`. Instead `"Hello"` unifies with the symbol `'Hello'`. All predicates that might be expected to handle strings (for example, `concat`), handle quoted symbols instead.

Note that the escape character for quoted symbols is the backslash character `"\"`. Thus the quoted symbol `'\\='` unifies with the unquoted symbol `\=`.

Input and Output

All the Edinburgh I/O predicates such as `see`, `seeing`, `seen` and `tell`, `telling` and `told` are not supported. To some extent they are subsumed by BNR Prolog's stream based I/O that use explicit stream identifiers such as `open`, `close` and `stream`.

Assert and Retract

The predicate `assert` in BNR Prolog has the same effect as `asserta` (rather than `assertz`). The assertion and retraction of clauses is possible only in the top-most context. If only `consult` and `reconsult` are used to add clauses to the clause space, this should make no difference to the behavior of programs that use `assert` and `retract`. However, if these same programs are loaded into contexts, they may behave differently.

All Solutions Predicates

The predicates `setof` and `bagof` are not supported. Instead the predicates `findset` and `findall` are provided.

Database Predicates

The database predicates `recorda`, `recordz`, `recorded` and `erase` are not supported. Instead their functions can be performed with the state space predicates `remembera`, `rememberz`, `recall`, `recallz`, `forget`, `forget_all` and `update`.

Metalogical Predicates

The metalogical predicates `"=.."` and `functor` are unnecessary in BNR Prolog because structures can be built and decomposed with unification.

Compatibility File

The following predicates and operators, found in many Edinburgh Prologs are not supported in BNR Prolog but are defined in a loadable file titled `Edinburgh` found on the release disk.

Operators: `,` `\+` `not` `=..`

Other Predicates:

<code>abolish</code>	<code>arg</code>
<code>assert</code>	<code>atom</code>
<code>bagof</code>	<code>call</code>
<code>clause/2</code>	<code>functor</code>
<code>get</code>	<code>get0</code>
<code>name</code>	<code>put</code>
<code>setof</code>	<code>skip</code>
<code>tab</code>	

Index

; 28
! (Edinburgh cut) 27
->; 30
-> (if-then) 29
{ } (constraints) 31
* (multiplication) 68, 75
** (exponentiation) 75
+ (addition) 68, 75
- (subtraction) 68, 75
/ (division) 68, 75
// (integer division) 68, 75

A

abort 170
accumulate 78
activewindow 233
acyclic 50
additem 332
addmenu 335
addresitems 336
addresmenu 337
ancestor goals 171
angle 278
Anonymous variables 15
Applications
 definition of 212
arcabs 299
arcrel 299

arg 59
Arithmetic
 comparisons 70
 functional 68
 operators 75
 relational 73
Arity 17
assert 122
asserta 122
assertz 123
at 94
atomic 48
attachpicture 317

B

backcolor 279
backpat 280
beep 368
beginpicture 318
bind_vars 60
block 33
break 170, 188
bucket 48
Buckets
 definition of 15
 print form of 15

C

Call port 167
caller 189
changedtext 247
Characters
 escape notation for 10
 special 10
circleabs 301
circlerel 301
clause 125
clause_head 126
Clauses 22
 body of 22
 examples of 23
 head of 22
 scope of 120
 syntax of 22
close 95
close_definition 127
closed_definition 128
closewindow 234
compound 48
concat 82
configuration 202
confirm 354
Constraints 31
 syntax of 17
consult 129
Contexts 119, 130
 syntax of 7
continue 190
Control
 predicates 26

copyfile 217
count 34
creep 170, 172
csize 259
cut 35
 ancestral 35
 Edinburgh 27

D

debug 175
Debugger 167, 172
 box model 167
 call port 167
 debug mode 172
 exit port 167
 fail port 168
 leashing 173
 port commands 170
 port messages 168
 redo port 167
 skipping 173
 spying 173
 writing a 185
decompose 61
defaultdir 218
defexternal 384
definition 131
delay 204
deletefile 219
deleteitem 338
deletemenu 339
deletepicture 319
deskaccessory 369
detachpicture 320
Dialogs
 boxes 353

- modal 353
- modeless 353
- predefined 353

dograf 254
dotext 248
doubletime 370
dragwindow 235

E

edit 272
enable_trace 191
endpicture 321
enter_context 132
Errors

- input/output 395
- messages 391
- run 391
- syntax 91, 393

Events

- detecting 346
- idle 345
- keyboard 345
- menuselect 349
- mouse 345
- reporting priority of 346
- types 345
- useractivate 348
- userclose 349
- userdeactivate 348
- userdownidle 350
- userdrag 349
- usergrow 349
- userkey 350
- usermousedown 349
- usermouseup 349
- userupdate 350

- userupidle 350
- userzoom 349
- window 345

Exit port 167
exit_context 133
External procedures

- data types 385
- parameter interface 386
- restrictions 387

F

Fail port 168
fail 36, 170
failexit 37
Files

- creator 215
- file types 215
- filenames 215
- full filenames 215
- partial filenames 215

fillpat 282
Filters

- basic 47

findall 38
findset 39
float 48
Floating point numbers

- input format of 13
- output format of 13
- standard order 53

foreach 40
forecolor 283
forget 151
forget_all 153
freeze 41

Full filenames 215

fullfilename 220

Functional arithmetic 68

Functors 17

G

get_char 96

get_term 97

getappfiles 371

Global state space 149

goal 192

grand_caller 193

Graphics

- absolute output descriptors
297

- attribute descriptors 253, 275,
276

- output descriptors 253, 275,
276

- relative output descriptors
297

- structures 253, 275

- turtle 296

ground 51

growwindow 236

H

halt 205

help 171, 172

hide 134

hidewindow 238

homedir 221

I

iconabs 302

iconrel 302

if-then 29

if-then-else 30

Indefinite lists 16

Input/Output

- character 92

- default streams 90

- errors 90

- failure conditions 90

- pipes 89

- streams 89

- system errors 395

- term 92

inqgraf 256

inqtext 249

insert 273

integer 48

integer_range 72

Integers

- examples of 13

- syntax of 13

interval 48

Intervals

- arithmetic functions 76

- arithmetic operations on

- intervals 75

- creating 73

- creating an interval 74
- definition of 14
- print form of 14
- standard order 53

invalidrect 372

inventory 154

is (arithmetic evaluation) 71

isdirectory 222

isfile 223

isfont 373

ispicture 322

isvolume 224

iswindow 239

K

Knowledge base 119

L

lasteventdata 347

lastmenudata 340

leap 171

leash 176

Leashing 173

line 303

lineabs 303

linere1 303

list 48

listdirectories 225

listener 194

listfiles 226

listfonts 375

listing 135

listpictures 323

Lists

- empty 16
- examples of 16
- indefinite 16
- standard order 53
- syntax of 16

listvolumes 227

listwindows 240

load_context 137

load_state 155

loadpicture 324

Local state space 149

localglobal 376

lowercase 83

lsize 260

M

Macintosh file system access 215

mbarheight 377

memory_status 206

menuItem 341

Menus

- creating 330
- menu bar 329
- resource 330
- system supplied 329

message 356

messagebutton 378

Metapredicates 58

mod 68

move 304

moveabs 304

moverel 304

N

name 84
namefile 357
namelength 85
new_state 156
nl 99
nodebug 171, 177
nonvar 48
nospy 172, 178
not 42
notrace 180
number 48
numeric 48

O

once 43
open 100
openwindow 241

Operators

- associativity of 18
- list of predefined 21
- name of 18
- precedence of 18
- term comparison 54
- type definitions of 19
- type of 18

or 28

ovalabs 305

ovalrel 305

P

Partial filenames 215

penmode 284

penpat 286

pensize 288

pictabs 307

pictrel 307

pictoscrap 325

Pictures

- as resources 315

- creating 315

- frame of 315

- ID numbers 315

- manipulating 315, 316

- Quickdraw 315

Pipes 89

polygon 308

popupmenu 343

Ports 168

position 289

positionwindow 243

predicate 138

Predicates

- definition of 23

- description of layout 2

- hidden 121

- local 120

- variadic 23

print 102, 171
print_interval 103
printfile 228
put_char 104
put_term 105

Q

query 359
quit 207

R

range 74
read 106
readln 107
recall 157
recallz 158
reconsult 139
rectabs 309
rectrel 309
Redo port 167
region 312
Relational arithmetic 73
reload_context 140
reloadtext 250
remember 159
remembera 159
rememberz 161
renamefile 229
repeat 44
replace 274

restart 208
retargettext 251
retract 141
retractall 143
retry 172, 195
rrectabs 310
rrectrel 310
Run errors 391

S

save_state 163
save_ws 144
savepicture 326
savetext 252
scale 290
scandirection 261
scrapcontents 379
scraptopict 327
scrndimensions 380
seek 108
select 360
selectafile 362
selectcabs 262
selectcrel 264
selection 266
selectlabs 268
selectlrel 270
selectone 364
Sentences
 syntax of 7

- Separators 7
 - set_end_of_file 109
 - set_trace 196
 - setcursor 381
 - sizewindow 244
 - skip 172
 - Skipping 173
 - solve 79
 - sort 56
 - spanning_tree 63
 - spied 181
 - spy 172
 - Spy points 173
 - spyall 183
 - Spying 173
 - sread 110
 - Standard order 53
 - State space 149
 - global 149
 - local 149
 - stats 209
 - stream 111
 - Stream pointer 89
 - Streams 89
 - default 90
 - Structures 48
 - argument of 17
 - arity of 17
 - cyclic 61
 - examples of 17
 - principal functor of 17
 - syntax of 17
 - substring 86
 - subsumes 64
 - swrite 112
 - swriteq 114
 - symbol 48
 - symbol_name 145
 - Symbols
 - examples of 10
 - Manipulating 81
 - standard order 53
 - syntax of 10
 - Syntactic context 7
 - Syntax errors 393
 - System
 - date 210
 - predicates 201
 - time 210
 - utility predicates 367
- ## T
- Tail variables
 - standard order 53
 - syntax of 16
 - tailvar 52
 - Term comparison operators 54
 - term_compare 57
 - termlength 65
 - Terms
 - basic 9
 - comparing 53
 - compound 9
 - definition of 9
 - standard order of 53

Text

- absolute position of 257
- attribute descriptors 246, 257, 258
- descriptors 246, 257
- output descriptors 246, 257, 271
- predefined style
 - characteristics 291
 - relative position of 257
 - selecting 258
 - structures 246, 257

textabs 313

textface 291

textfont 292

textmode 293

textrel 313

textsize 294

timedate 210

trace 184

traceback 197

tracer 199

Transfer modes 284

true 45

try 198

turn 314

Turtle graphics 296

U

update 164

uppercase 87

userevent 348

userpat 295

V

validrect 382

var 48

Variable names 15

Variables 66

- anonymous 15

- canonical name of 60

- examples of 15

- name preservation of 15

- scope of 15

- standard order 53

- syntax of 15

Variadic predicates

- example of 23

version 211

visible 146

W

Windows

- graphics 231

- manipulating graphics 253

- manipulating text 246

- names of 231

- of type graf 231

- of type text 231

- types of 231

with_context 147

World stack 119

write 116

writeq 117

Z

zoomwindow 245

© Bell-Northern Research Ltd. 1988

All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language by any means without the written permission of Bell-Northern Research Ltd.

Printed in Canada

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100