8.1

Programming Guide

by Rebecca Shalfield

WIN-PROLOG 8.1

The contents of this manual describe the product, **BDS-PROLOG** for Windows (hereinafter called **WIN-PROLOG**) and one or more of its LPA Toolkits, and are believed correct at the time of going to press. They do not embody a commitment on the part of Logic Programming Associates (LPA), who may from time to time make changes to the specification of the product, in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose without the prior written agreement of LPA.

Authors: Rebecca Shalfield, Clive Spenser, Brian D Steel and Alan Westwood

Logic Programming Associates Ltd
PO Box 226
Cranleigh
Surrey
GU6 9DL
England

phone: *+44 (0) 20 8871 2016*

web site: *http://www.lpai.uk*

01 May 2025

Contents

Programming Guide Contents

List of Tables

Introduction

Welcome to *WIN*-PROLOG: a Prolog designed to run on the Windows platform with the ability to access as much memory as is available on your machine. *WIN*-PROLOG is a flexible system, providing all the features expected of a general purpose programming language. With its fast incremental compiler *WIN*-PROLOG allows you to develop Prolog programs interactively. This interactive mode of development enables the rapid prototyping of Prolog applications. *WIN*-PROLOG also offers an optimising compiler that generates compact and efficient object code.

This manual is a companion to the 'Technical Reference'. Outlined here are the logical groupings and subjects of the *WIN*-PROLOG predicates (documented in the 'Technical Reference' in alphabetical order). The manuals provided with *WIN*-PROLOG do not try to teach you how to program in Prolog. If you would like to learn more about Prolog, then the books mentioned in the reference section at the end of this chapter may prove helpful.

Features of *WIN*-PROLOG

The syntax supported by *WIN*-PROLOG is the industry standard "Edinburgh Syntax" (also known as the DEC-10 syntax). In addition *WIN*-PROLOG provides other built-in predicates which support:

- Quintus Prolog Compatibility.
- Double precision floating point arithmetic.
- Formatted input and output.
- Logical file handling.
- List sorting, concatenation and membership.
- Definite Clause Grammars (DCG).
- Graphics handling.
- Extensive Operating System interfaces (DOS/Windows/Macintosh).
- Graphical User Interface (GUI) handling (Windows/Macintosh).
- Memory files.
- Calling Windows system API or 32-bit DLL functions.
- Encryption.
- Data compression.

Notation Conventions

Predicate Definitions

When predicate definitions are given, the functor, arguments and positions of the arguments of the predicate are shown as a template such as:

foo(*+Arg1*, *?Arg2*, *-Arg3*)

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

Mode Declarations

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

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

Table 1 - mode declaration symbols

Prolog Listings

Listings of Prolog programs and examples of Prolog queries are shown in 'Courier New' font. The text that you actually type in is shown in **bold**. Text that is output by *WIN-*PROLOG and supplementary comments are shown in plain text.

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

Horizontal ellipses (...) are used as a shorthand in examples to indicate that any number of items may be entered.

foo(arg_1, arg_2, ..., arg_n) ($n < 1$)

denotes a compound term with at least one argument.

Argument References

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

Tables of Information

Tables of information are shown in 'GillSans' font.

Predicate References

References to predicates in the main text appear italicised and are generally given in *functor/arity* form, such as: *foo/2*.

References

K.L.Clark and F.G.McCabe. *micro-PROLOG: Programming in Logic*. Prentice-Hall International, 1984. (This book does not describe the Edinburgh syntax.)

W.F.Clocksin and C.S.Mellish. *Programming in Prolog*. Springer Verlag.

I.Bratko. *Prolog Programming For Artificial Intelligence*. Addison-Wesley Publishing Company.

R.A.Kowalski. *Logic For Problem Solving*. Artificial Intelligence series. North Holland Inc.

T.Dodd. *Prolog: A Logical Approach*. Oxford University Press

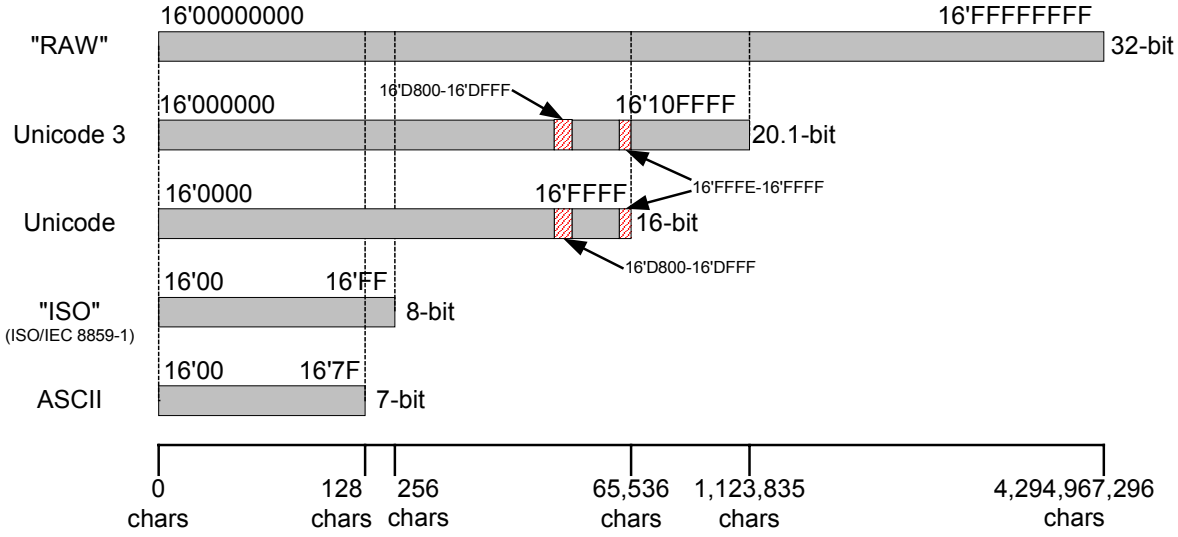
Logic Programming Associates Ltd do not endorse these books or recommend them over others on the same subject.

Syntax

In this chapter we describe the Edinburgh syntax of ***WIN-PROLOG*** . This syntax is essentially the syntax of the book - *Programming in Prolog* by W.F.Clocksin and C.S.Mellish (published by Springer-Verlag).

Character Set

WIN-PROLOG has full "Unicode" support. Internally, ***WIN-PROLOG*** can handle any 32-bit character code. All the characters with special meaning to ***WIN-PROLOG*** syntax are confined to the first 128 characters (the 7-bit ASCII character set). The characters with character codes above 127 contain international characters, accents, and graphics characters.



Separators and Terminators

The normal term separator is the comma. This is used to separate terms in lists and argument lists. A space must be used to separate an operator from an operand if they are both of the same token type (e.g. they are both alphanumeric tokens).

The usual term terminator is the full stop followed by a space or carriage return. (A full stop not followed by a space or carriage return is treated as a symbolic atom - see below.)

Note that a space between an atom and a left parenthesis, '(', is significant.

Comments

Comments have no effect on the behaviour of a program. In fact they are ignored when a Prolog term or program is read in. There are two forms of comment:

1. A sequence of characters that begins with the symbol "/*" and ends with "*/" is treated as a comment.

2. A sequence of characters that begins with the symbol "%" and ends with the end-of-line character (carriage return) is treated as a comment.

The first type of comment allows a comment to extend over several lines. The second type of comment is useful when commenting a single line. For example:

```
/*
this is a comment
*/
% so is this
```

Terms

Terms are the fundamental data types of ***WIN-PROLOG***. They are the building blocks from which Prolog clauses, and commands are constructed.

Here we describe the basic term types and their syntax: variable names, integers, floating point numbers, atoms, strings (a unique ***WIN-PROLOG*** text data type) and compound terms (i.e. lists and char lists (normally called strings in 'Edinburgh' parlance)).

Variable Names

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

```
Anything    _var        _1          X           Var1
```

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

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

An underscore on its own is an anonymous variable.

Integers

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

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

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

Floating Point Numbers

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

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

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

The following are *not* floating point numbers:

```
.9          % does not start with a digit

3e-22       % no decimal point

34.1 e3     % contains a space before the 'e'

-.7         % no digit after the minus sign

56.1e4.8    % exponent is not an integer

23.         % no digit after the decimal point
```

Number Bases

You can enter numbers in a particular base in ***WIN-PROLOG*** using the ' notation. As in the following examples:

```
?- X = 16'F.            % a number in base 16
X = 15

?- ?- X = 2'10010101. % a number in base 2
X = 149
```

Incidentally you can also use this notation to give the character code of the character following the quote sign, as in the following:

```
?- X = 0'a.
X = 97
```

Atoms

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

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

Alphanumeric Atoms

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

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

```
apple       a1          apple_cart    test_1_case
fool23      f_T1        fred          longTable
```

Symbolic Atoms

A symbolic atom is written as a sequence of symbolic characters, and characters with character codes above 7Fh (127). The symbolic characters are shown in *Table 2 - symbolic atoms*:

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

Table 2 - symbolic atoms

The following are all symbolic atoms:

```
&      &:     ++     <<     >>     <-     ..     *-/*
```

Note that the /* appearing in the last example is not interpreted as the start of a comment.

Quoted Atoms

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

```
''
```

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

```
'~I'
```

represents ctrl-I.

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

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

Examples

```
'Apple'          '123'        '~(0)'          'hello world'

'~Ibold~M~J'     '~(41)'      '~(FFFFffff)'   'don''t care'
```

The last example represents the atom:

```
don't care
```

Note that '~F' is not the same as '~(F)'.

Special Atoms

The special atoms are shown in the following table:

```
!       ;
```

Table 3 - special atoms

Please note that the special "*empty list*" atom, "*[]*", is not a true atom in ***WIN-PROLOG***.

Strings

The string is a text data type specific to ***WIN-PROLOG***. The maximum length of a string is approximately 3 gigabytes. A string is any sequence of characters surrounded by backwards quotes.

To insert a backwards quote character in a string use two adjacent backwards quote characters:

```
``
```

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

```
`~I`
```

represents ctrl-I.

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

Examples

```
`Apple`          `123`       `~(0)`          `hello world`

`~Ibold~M~J`     `~(41)`     `~(FFFFffff)`   `don``t care`
```

The last example represents the string:

```
don`t care
```

Note that `~F` is not the same as `~(F)`.

Compound Terms

In *WIN-PROLOG* tuples, lists, char lists, conjunctions and disjunctions are compound terms.

Tuples

A tuple (which is also a compound term) is a structured data item that consists of a functor followed by a sequence of one or more arguments which are enclosed in brackets and separated by commas. The general form of a tuple is:

$$functor(t_1, t_2, \ldots, t_n) \qquad n \geq 1$$

functor is the functor. It can be an atom or a variable name. (For further details about the use of a variable name as the functor please see the section below entitled "Meta-variables").

The term t_i represents the *i*'th argument of the tuple.

The arity of a tuple is the number of arguments it has (*n* in the example above). We refer to functor with arity *n* using the notation:

functor/n

The following are examples of tuples:

```
likes(paul,prolog)      % functor is likes (arity is 2)
read(X)                 % functor is read (arity is 1)
>(3,2)                  % functor is > (arity is 2)
```

A tuple can be thought of as representing a record structure. The functor represents the name of the record, while the arguments represent the record fields.

Certain functors can be written as operators. For more information see the section below entitled "Operators". Note: There must be no space between the functor and the opening parenthesis of a tuple. For example:

```
likes (paul,prolog)
```

is <u>not</u> a legal tuple. Spaces between the arguments are allowed however:

```
likes( paul, prolog )
```

Lists

A list (which is also a compound term) is a sequence of terms of the form:

$[t_1,t_2\ldots,t_n]$ $n \geq 0$

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

```
[]
```

The following example is a four element list:

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

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

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

We also represent a list using the notation:

[*t1*, *t2*, ..., *ti* | *Variable*] $i \geq 1$

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

For example the list pattern:

```
[Head|Tail]
```

could be unified with the list:

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

to give the variable bindings:

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

Conjunctions

A conjunction (which is also a compound term) is a sequence of terms of the form:

$(t_1,t_2\ldots,t_n)$ $n \geq 0$

The term t_i is the *i*'th element of the conjunction.

Disjunctions

A disjunction (which is also a compound term) is a sequence of terms of the form:

$$(t_1|t_2|\ldots|t_n) \qquad n \geq 0$$

The term t_i is the i'th element of the disjunction.

Char Lists

A char list (which is also a compound term) is a sequence of characters surrounded by the double quotes character ("). It is simply an abbreviation for the list of decimal integer character codes of the characters in the sequence. For example, the char list:

```
"A boy"
```

is simply a shorthand form of the list:

```
[65,32,98,111,121]
```

```
?- X = "A boy". <return>
X = [65,32,98,111,121]
```

To insert a double quote character in a char list use two adjacent double quote characters:

```
""
```

As with quoted atoms the tilde character is used as an escape character, allowing you to enter control characters in a char list. For example:

```
"~G"
```

represents the list:

```
[7]
```

(To insert a tilde in a char list use ~~.)

```
?- X = "A "" ~G ~~ boy". <return>
X = [65,32,34,32,7,32,126,32,98,111,121]
```

Operators

Operators allow you to use an alternative syntax for compound terms. There are three types of operator: prefix, postfix and infix.

Prefix Operators

The compound term:

functor(*term*)

can also be written as:

functor term

if *functor* has been declared a prefix operator. For example, the built-in predicate *spy/1* is a prefix operator which means that the following compound term can be entered:

```
spy my_module
```

This is equivalent to:

```
spy(my_module)
```

Postfix Operators

The compound term:

functor(*term*)

can also be written as:

term functor

if *functor* has been declared a postfix operator. For example, if is_male/1 has been declared a postfix operator then you could enter the compound term:

```
paul is_male
```

This is equivalent to:

```
is_male(paul)
```

Infix Operators

The compound term:

functor(*term*$_1$,*term*$_2$)

can also be written as:

term$_1$ *functor term*$_2$

if *functor* has been declared an infix operator. For example, the built-in arithmetic function '+' is an infix operator which means you can enter the compound term:

```
5 + 10
```

This is equivalent to the predicate:

```
+(5,10)
```

Note: in ***WIN-PROLOG*** there is no definition for the +/2 predicate (+ is simply an argument given to the is/2 predicate). so entering this at the command line will give:

```
| ?- 5 + 10 .
! ----------
! Error 20: Predicate Not Defined
! Goal   : 5 + 10
```

Operator Precedence

Every operator has a precedence associated with it. This is an integer between 1 and 1200. It is used to disambiguate expressions that contain several operators. The lower the precedence, the more strongly an operator binds to its arguments.

For example, the expression:

```
2 + 5 * 8
```

represents the term:

```
+(2,*(5,8))
```

Because '*' (whose precedence is 400) binds more strongly than '+' (precedence is 500). (Note that operators with a higher precedence appear at a higher level of the compound term than lower precedence operators.) .

Operator Types

The type of an operator defines its associativity. It is used to disambiguate an expression that contains two operators of the same precedence. If an operator is non-associative then its arguments must be sub-expressions of strictly *lower* precedence than the operator itself.

A left associative operator is one whose left hand argument may be a sub-expression of the same precedence as the operator itself (it can also be lower). A right associative operator is one whose right hand argument may of the same (or lower) precedence as the operator. For example, the built-in operators '+' and '-' are both left associative infix operators with a precedence of 500. This means that the expression:

```
10-5+2
```

represents the compound term:

```
+(-(10,5),2)
```

because the left hand argument of '+' can have the same precedence. The '-' operator *cannot* have a right argument with the same precedence. This means that the following compound term is *not* a valid interpretation of the above expression:

```
-(10,+(5,2))
```

because the right hand argument would have the same precedence as '-' itself (and '-' is not right associative). The various types are shown in *Table 4 - operator types and meanings*.

Operator Type	Meaning
fx	non-associative prefix operator
fy	right associative prefix operator
xf	non-associative postfix operator
yf	left associative postfix operator
xfx	non-associative infix operator
xfy	right associative infix operator
yfx	left associative infix operator

Table 4 - operator types and meanings

Note that these types indicate the associativity and position of an operator.

Declaring Operators

Operators are declared using the built-in predicate *op/3*. The form of this predicate is:

op(*+Precedence*, *+Type*, *+Name*)

where *Precedence* is the operator's precedence (an integer in the range 1 to 1200), *Type* defines the operator type and associativity (e.g. fx), and *Name* is the name of the operator (or a list of operator names). If *Precedence* is 0 then the operator declaration for *Name* is cancelled.

Examples

The following examples show how some of the built-in operators are defined.

```
op(200, xfy, ^).
op(500, fx, [+, -]).
```

It is possible to have more than one operator of the same name. For example, the built-in operator '+' is declared as both a prefix and an infix operator. The built-in predicate *current_op/3* can be used to find out what operators are currently defined. The format of this predicate is:

current_op(*?Precedence*, *?Type*, *?Name*)

This succeeds if there is an operator called *Name* of type *Type* and with a precedence of *Precedence*. It can be used to backtrack through the list of currently defined operators.

Examples

```
current_op(X, Y, Z).
current_op(500, X, Y).
```

Program Structure

In this section we describe the syntax of ***WIN-PROLOG*** programs. A Prolog program is made up of one or more of the following program elements:

- clauses
- grammar rules
- commands

We describe the format of these program elements in turn.

Clauses

Clauses are the building blocks of Prolog programs. There are two types of clause: facts and rules.

A fact is of the form:

head.

where *head* is the head of the clause. *head* may be an atom or a compound term whose functor is any atom except :-. A fact is terminated by a '.' followed by a white space character (e.g. a space, or a carriage return).

A rule is of the form:

$$head :- t_1, t_2, \ldots, t_k \qquad (k \geq 1)$$

where *head* is the head of the clause and the terms to the right of ":-" are the body of the clause. Each t_k is known as a call term or goal. A call term must be an atom (a 0-argument call), a compound term, or a variable name. A rule is terminated by a '.' followed by a white space character.

The functor of the head of a clause is the predicate that the clause describes. All the clauses describing a given predicate comprise its definition. The arity of a clause is the number of arguments in its head.

Examples

```
foo.                                    % a fact

foo(1) :-
        bar.                            % a rule

likes(Anyone, prolog) :-                % a rule
        logic_programmer(Anyone).

likes(Anyone, Anything).                % a fact

my_append([], X, X).                    % a fact

my_append([A|B], C, [A|D]) :-           % a rule
        my_append(B, C, D).

is_not_true(X) :-
        X,
        !,
        fail.

is_not_true(X).
```

These clauses define the relations foo/0, foo/1, likes/2, my_append/3 and is_not_true/1 respectively.

All clauses describing a predicate must be in a single source file unless the predicate is declared as multifile (see multifile/1).

Grammar Rules

A Prolog program may contain one or more grammar rules. These grammar rules may be used to define the syntax of a language and to define a parser for that language.

A grammar rule takes the form:

grammar_head - -> *grammar_body*.

Where *grammar_head* is a non-terminal symbol optionally followed by a terminal symbol. The body of the grammar rule is a sequence of terminals, non-terminals or grammar conditions, each separated by commas or semi-colons. A grammar condition is a sequence of Prolog call terms surrounded by curly brackets ('{' and '}').

For a detailed description of the Prolog grammar rules please see the chapter on 'Grammar Rules'.

Examples

```
sentence - -> noun_ph, verb_ph.
verb_ph - -> verb, noun_ph.
verb - -> [likes] ; [hates].
noun_ph - -> determiner, noun.
determiner - -> [the].
noun - -> [boy] ; [dog].
```

Commands

A command is a Prolog term of the form:

:- $goal_1$, ..., $goal_k$, % k ≥ 1

where $goal_i$ is a call term (i.e. goal). A command is executed automatically when it is encountered during a consult or reconsult (see also *initialization/1*).

Example

```
:- write('hello world'),nl.
```

Meta-variables

A meta-variable is a variable which appears in place of a callable Prolog structure. Two types of meta-variable are allowed in ***WIN-PROLOG***: condition meta-variables and predicate meta-variables.

Extended Meta-variable Facilities in *WIN-PROLOG*

The meta-variable facilities of ***WIN-PROLOG*** extend the usual Edinburgh syntax in two ways:

- condition meta-variables can be bound to atoms and compound terms.
- predicate meta-variables can be used (standard 'Edinburgh' syntax does not allow this).

Condition Meta-variable

This is where a variable appears as a goal in the body of a rule. The head of a clause may not be represented in this way. By the time the meta-variable is called it must have been instantiated to one of the following:

- an atom (represents a call to a 0-argument relation).
- a compound term of the form:

$$relation(t_1, t_2, \ldots, t_n)$$

The effect of evaluating a condition meta-variable is the same as if the condition had appeared in the source program instead of the meta-variable.

Examples

The following is the definition of the built-in predicate *\+/1*:

```
\+(X) :- X, !, fail.
\+(X).
```

You can query this predicate as follows:

```
\+(true).
no

\+(false)
yes

\+(compare(=, 2, 3)).
yes
```

Predicate Meta-variable

This is where a goal in the body of a rule is a compound term whose functor is a variable. By the time the goal is evaluated, the meta-variable must have been bound to an atom.

Examples

```
map(Pred, [], []).
map(Pred, [X|Y], [X1|Y1]) :-
        Pred(X, X1),
        map(Pred, Y, Y1).
```

In this example, it is assumed that the meta-variable 'Pred' will be bound to the name of a binary relation. Given the following binary relations:

```
double(X, Y) :-
        Y is X + X.
```

```
square(X, Y) :-
        Y is X * X.
```

You can query map/3 as follows:

```
map(double, [1,2,3,4], X).
X = [2,4,6,8]

map(square, [1,2,3,4], X).
X = [1,4,9,16]
```

Note: In standard Edinburgh syntax, the call to 'Pred(X,X1)' in the second clause for 'map/3' would have to be replaced with calls to *=../2* and *call/1* as follows:

```
map(Pred, [X|Y], [X1|Y1]) :-
        Call =.. [Pred,X,X1],
        call(Call)
        map(Pred, Y, Y1).
```

WIN-PROLOG supports *=../2* and *call/1*, but the previous method for meta-calling is far more efficient.

Arithmetic

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

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

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

Rounding errors will invariably occur during certain operations because many decimal fractions have no direct binary representation. These errors are normally confined to the 14th or 15th digit. No attempt is made to round results to fewer decimal places. For example, if the result of a calculation is the value 1.9999999999997 this value would not be converted to the integer 2; however there are functions to perform such rounding explicitly.

Predicates Related to Arithmetic

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

Arithmetic Expressions

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

An arithmetic expression can be one of the following:

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

Examples

The following are all legal arithmetic expressions.

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

Table 5 to *Table 10* outline the arithmetic functions that can be used with the *is/2* predicate.

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

Table 5 - basic arithmetic functions

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

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

Table 6 - trigonometric functions

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

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

Table 7- logarithmic functions

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

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

Table 8- truncation functions

The following bitwise operators will only work on integer values and will generate an error if any other type of input is given (including floating point numbers).

Function	Description
X /\ *Y*	the logical and of the integers *X* and *Y*.
X \/ *Y*	the logical inclusive or of the integers *X* and *Y*.
X << *Y*	the logical shift arithmetic left of the integer *X* by the number *Y* bits (vacated bits are filled with zeros).
X >> *Y*	the logical shift arithmetic right of the integer *X* by the number *Y* bits (the most significant bit is propagated into the vacated bits).
\(*X*)	the logical negation of the integer *X*.
a(*X*,*Y*)	the logical 'and' (AND) of the integers *X* and *Y*.
l(*X*,*Y*)	the logical left rotation of the integer *X* by the number *Y* bits.
o(*X*,*Y*)	the logical inclusive 'or' (OR) of the integers *X* and *Y*.
r(*X*,*Y*)	the logical right rotation of the integer *X* by the number *Y* bits.
x(*X*,*Y*)	the logical exclusive 'or'(XOR) of the integers *X* and *Y*.

Table 9 - is/2 integer bitwise arithmetic functions

Pseudo Random Number Generator

The pseudo random number generator (see *Table 10*) returns numbers that will be useful in simulations and games. The pseudo random sequence has a cycle of 2^64 numbers. Each time Prolog is invoked, the seed used by the random number generator is initialised from a combination of time and date. This ensures different behaviour on different occasions.

Function	Description
rand(*X*)	a random floating point number between zero and *X*.

Table 10 - random number function

The seed can also be initialised using the *seed/1* predicate. This allows the same sequence of random integers to be generated on different occasions (useful for some simulations, and for testing).

A few words of background should be said about "random" numbers and their use, as they can be of critical importance to simulations and related applications. Firstly, it should be noted that it is *not* possible to generate true random number sequences on standard Windows-compatible computer hardware, since this hardware is entirely deterministic. The best that can be achieved is to generate *pseudo* random numbers, hence the term, "Pseudo Random Number Generator", or "Prang". Using a Prang would have one major advantage over true random numbers, even if the latter were possible on a computer, namely that a given "random" sequence can be repeated where necessary to eliminate the effects of randomness from successive runs of a simulation.

The Linear Congruential Method

Many ways have been investigated of producing pseudo random number sequences on determistic hardware, and most involve complex shifting and indexing on tables of "seed" numbers. The careful choice of the initial seeds is essential to the good "random" behaviour of the Prang. Many such methods lack theoretical tests to support their claims to randomness, and so have been avoided in ***WIN-PROLOG***.

One method which involves only one seed, and which is supported by considerable amounts of theory, is the widely-used "linear congruential" generator. In such a prang, each successive pseudo random number is obtained by multiplying its immediate predecessor (the seed) by a carefully chosen multiplier, adding an equally chosen increment, and then returning the result modulo a suitable large number. The quality of such sequences is entirely dependent upon the correct choice of multiplier, increment and modulus: the worst case can yield a linear sequence; the best can yield pseudo random sequences of the highest quality.

The multiplier, increment and modulus used in ***WIN-PROLOG*** are chosen according to the criteria described in Vol 2 of "The Art of Computer Programming", by D E Knuth, and pass all known tests for randomness (including the "spectral" test) with flying colours. The generator yields a very random-appearing cycle of 2^64 distinct 64-bit numbers.

Seeding the Prang

When ***WIN-PROLOG*** starts up, one of its operations is to obtain the 64-bit value in a pair of low level system timers, and to store the result in the Prang seed. This ensures that ***WIN-PROLOG***'s Prang will yield a different sequence of pseudo random numbers every time it is run. As was noted above, in simulations it is often desirable to re-run a pseudo random sequence on different simulation models, to remove any bias that may

be due to the random numbers themselves, and for this purpose, ***WIN-PROLOG*** includes a *seed/1* predicate. This may be called with any 32-bit integer or 64-bit floating point number as its argument: the number will define a point in the cycle of 2^64 random numbers from which to resume generation.

Randomising the Prang

If it is desired to "randomise" the number sequence, perhaps after an explicit seed has been set, and a simulation performed, the simplest method (equivalent to that used during the ***WIN-PROLOG*** startup sequence) is to invoke the *time/2* predicate to return a value from the system timer, and to pass the result directly into the *seed/1* predicate, as the following program suggests:

```
randomise :-
        time( 1, X ),
        seed( X ).
```

It is important to note that "randomising" during a pseudo random number sequence can *degrade* the quality of the sequence. The obvious reason is that numbers may become related to time (because the seed is generated from a clock). Less obvious perhaps is that the Prang in ***WIN-PROLOG*** has been chosen because it passes all theoretical tests with flying colours, and is fast, and has shown itself to work very well. Randomising during a sequence violates the theory behind many of the tests, resulting in a loss of confidence in the sequence's pseudo random properties. The randomise program shown above should only be used to restart the Prang after an explicit seed has been set for a previous run of a simulation.

Timing

WIN-PROLOG has several predicates that relate to the system time and date and to the timing of programs. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Timing

ms/2	*call a goal and return its execution duration*
time/2	*get elapsed running time or local computer time*
time/4	*convert between day number and date*
time/5	*convert between tick count and time*
time/7	*return the local machine date and time*

Getting the System Date

The *time/2* predicate can be used to return the system date. As an example of its use, the following program uses *time/2* in conjunction with *stamp/2* to set the timestamp of the file, "foo":

```
?- time( 1, T ), stamp( foo, T ). <enter>
T = (145822,36522000).
```

Timing Programs and Time Stamps

If you want to time some Prolog programs, to either benchmark the Prolog or to generate some statistics on your own programs, *WIN*-PROLOG has two built-in predicates that deal with high resolution timing.

ms/2 takes a Prolog goal as an argument, then runs it and finally returns the number of milliseconds it took to run.

time/2 returns the current value of the internal hardware clock counter.

Configuration Options

There are a number of configuration options that govern the way that ***WIN-PROLOG*** interacts with the user. The options include: the reporting of file errors, the type of source-file style checking, setting the Prolog prompt, checking the values of ***WIN-PROLOG*** command line switches, changing the default file extensions, changing the debugger, affecting garbage collection. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Configuration Options

fileerrors/0	*turn on the reporting of file error messages*
no_style_check/1	*turn off the specified style of compile-time style checking*
nofileerrors/0	*turn off the reporting of file error messages*
prolog_flag/2	*get or check the values for global environment variables*
prolog_flag/3	*set and get values for global environment variables*
prompt/2	*get or set the Prolog prompt*
prompts/2	*get or set the buffered console input prompts*
style_check/1	*turn on the specified type of compile-time style checking*
switch/2	*set or get the value of a **WIN-PROLOG** command line switch*

Turning Style Checking On and Off

When a source code file is loaded into ***WIN-PROLOG*** you have the option of using a built-in style checker to check different aspects of your Prolog programming style. In themselves the style warnings do not indicate errors but may point to potential bugs. The predicates used to turn style checking on or off are: *no_style_check*/1 and *style_check*/1. Style checking is described in more detail in the chapter on debugging.

Turning the Reporting of File Errors On and Off

When an attempt is made to read or write to a file and the file cannot be opened, the usual behaviour is for a file handling error to be invoked and the goal that is running to be aborted. This is not always convenient in a program, so the facility of turning the reporting of file errors off is provided. In this case when an attempt is made to read or write to a file and the file cannot be opened the goal that was running will simply fail. The predicates that perform this function are *fileerrors/0* and *nofileerrors/0*.

Changing or Getting the Prolog Read Prompt

The Prolog read prompt can be configured by the user using the goal *prompt/2*. The read prompt indicates that the Prolog system is waiting for user input. When the prompt is set it remains in force until control has been returned to the Prolog command line. An example of using *prompt/2* is when an information line is presented to the user to indicate the type of input expected.

```
?- prompt( P, '==>' ), read( T ), prompt( _, P ).        <enter>
==>new(prompt).                                          <enter>
P = '|: ' ,
T = new(prompt)
```

Retrieving or Setting a *WIN*-PROLOG Switch

***WIN*-PROLOG** has 26 built-in switches which can be used as simple semi-permanent integer storage spaces by the user. They are used initially to retrieve any command line switches that have been set but from then on may be used freely. The predicate used to set or retrieve the values is *switch/2*.

System flags

***WIN*-PROLOG** makes use of some flags that denote the defaults for the system; you can test these using a built-in predicate called:

```
prolog_flag(Flag, Value).
```

where *Flag* is the name of the flag to be checked and *Value* is the value of the flag. You can change the flags with a built-in predicate called:

```
prolog_flag(Flag, Oldvalue, Newvalue).
```

where *Flag* is the name of the flag to be changed, *Oldvalue* returns the old value of the flag and *Newvalue* is the value the flag is to be set to. Changing the Prolog flags effects the way the ***WIN*-PROLOG** environment works. The following query will set the status box to report on the compilation of files.

```
?- prolog_flag(status_box, Old, on).
```

so that whenever a file is optimized a status box will appear and display each predicate as it is optimized.

The names of some useful flags and the values that they can take are shown in the following tables:

Flag	Default Value	Other Values
text_extension	''	any <atom>
source_extension	'.PL'	any <atom>
object_extension	'.PC'	any <atom>
project_extension	'.PJ'	any <atom>
flex_extension	'.KSL'	any <atom>

Table 11 - prolog flags for defining file extensions

The above flags define the default extensions for the various types of files used in ***WIN-PROLOG***. They can be changed using *prolog_flag/3*, for example, to change the default extensions for ***WIN-PROLOG*** source files from '.PL' to '.PRO' you could use the following goal:

```
?- prolog_flag(source_extension,_,'.PRO').
```

Then any source files that are loaded by ***WIN-PROLOG*** should have the default extension '.PRO'.

Flag	Default Value	Other Values
debug_file	srcbug	<atom> in the domain {srcbug, boxbug, mismatch, failure, monitor, <atom>}

Table 12 - prolog flag for setting the debugger

The type of debugger can be set by changing the 'debug_file' flag with *prolog_flag/3* to one of the provided files or a user-defined debugger. For example, the following query will change the debugger from the default source-level debugger to the text based four-port box model debugger.

```
?- prolog_flag(debug_file, _, boxbug).
```

Flag	Default Value	Other Values
max_depth	0	<integer> ≥ 0

Table 13 - prolog flag for write_term/2

The 'max_depth' flag sets the output depth in *write_term/[2,3]* a library predicate provided for compatibility with Quintus Prolog. For a complete description of *write_term/[2,3]* see your Quintus Prolog documentation.

Flag	Default Value	Other Values
unknown	error	<atom> in the range {error, fail}

Table 14 - prolog flag for setting the system unknown predicate handling

The 'unknown' flag defines the system action taken when an undefined predicate is called. For example the following call will set the system to fail whenever an undefined predicate is called:

```
?- prolog_flag(unknown,_,fail).
```

Control

WIN-PROLOG provides a number of predicates that allow extra control over the execution of programs. Normally a Prolog program searches non-deterministically for a solution, backtracking on failure and terminating when the first solution has been found. The control predicates allow this behaviour to be modified to a greater or lesser extent. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Control

!/0	*control backtracking*
,/2	*conjunction*
->/2	*if then*
;/2	*disjunction*
\+/1	*negation as failure*
abort/0	*abort the current program*
break/0	*suspend the current execution*
break_hook/1	*built-in break hook*
exit/1	*exit directly to the operating system*
fail/0	*force failure*
false/0	*force failure*
halt/0	*terminate the current Prolog session*
halt/1	*terminate the current Prolog session with a return code*
not/1	*logical negation*
otherwise/0	*succeed.*
repeat/0	*succeed even on backtracking.*
repeat/1	*succeed even on backtracking for a given number of times*
true/0	*succeed*

Controlling Backtracking

The cut predicate is included to prevent unwanted backtracking into previous calls in the current clause, and to other clauses. Consider the following program:

```
a :- b, !, c.        % clause 1
a :- d.              % clause 2
```

When the cut in clause 1 is executed it will prevent backtracking into both goal "b" and clause 2. This effect can be used to limit backtracking through a database. For instance looking at the 'person' database from the negation example, you can limit backtracking to stop at 'dave':

```
person(clive).
person(dave):- !.
person(diane).
```

So the program:

```
write_person :- person(X), write(X), nl, fail.
```

Will output:

```
clive
dave
no
```

Placing a cut at the end of a clause will have the effect of making the procedure deterministic. For example:

```
a :- b, c.
b :- d, e, !.
b :- f.
```

Assuming the first clause for goal "b" is succesful, the cut at the end of the clause will force "b" to be deterministic (even though there is a second clause for "b")

The predicate *one/1* can be used instead of a cut to make a procedure call deterministic. For example:

```
a :- one(b), one(c), d.
```

The calls to "b" and "c" are deterministic in this clause.

Conjunction

The standard 'Edinburgh' way of signifying the conjunction (and) of goals is by using the comma. For example:

```
a :- b, c.
```

Means that for "a" to succeed both "b" *and* "c" must succeed. Either subgoal may itself be a conjunction or disjunction, or a simple goal.

Disjunction

The standard 'Edinburgh' way of signifying the disjunction (or) of goals is by using the semi-colon. For example:

```
a :- b; c.
```

Means that for "a" to succeed either or both of "b" and "c" must succeed. Either subgoal may itself be a conjunction or disjunction, or a simple goal.

If-Then

The standard 'Edinburgh' way of signifying implication (If-Then) is by using a combination of a dash and a right arrow. For example:

```
a :- b -> c.
a :- d.
```

Means that *if* "b" succeeds *then* try "c", *if* "c" then fails do not backtrack into "b", but go straight on to the next clause for "a". In this way 'If-Then' can be thought of as a local cut.

If-Then-Else

An implication (If-Then) can be combined with a disjunction (or) to form an If-Then-Else. For example:

```
a :- (b -> c; d), e.
```

Means that *if* "b" succeeds *then* try "c" followed by "e", *else* (*if* "b" fails) try "d" followed by "e".

Negation as Failure

Negation as failure, indicated using the '\+'/1 predicate, checks that a given goal does not succeed; this predicate succeeds if the given goal fails and fails if the given goal succeeds. For example:

```
a :- \+ b.
```

Means that "a" will succeed if "b" fails and vice-versa.

WIN-PROLOG also provides a 'logical' *not/1* predicate which also succeeds if the given goal fails. If any variables in the goal are instantiated as a result of running the goal, then *not/1* will generate an error. For example consider the database:

```
person(clive).
person(dave).
person(diane).
```

The query:

```
?- not person(clive).
```

fails because 'clive' is a person in our database. Whereas the query:

```
?- not person(xzzyblyk).
```

succeeds because 'xzzyblyk' is quite evidently not a person in our database. However, the query:

```
?- not person(Anything).
```

will generate an error because 'Anything' is a variable that will be instantiated to the first person found in the database. Whereas if *\+/1* was used instead of *not/1*:

```
?- \+ person(Anything).
```

this query will simply fail.

Forcing Failure

Predicates can be forced to fail and cause backtracking (the finding of alternative solutions) using the *fail/0* or the *false/0* predicates. These predicates are identical apart from their names and they always fail when they are called. For example, using the 'person' database from the previous example:

```
person(clive).
person(dave).
person(diane).
```

you can write a program that will print out all the names in the database using backtracking:

```
write_person :- person(X), write(X), nl, fail.
```

Then the query:

```
?- write_person.
```

will produce the following output:

```
clive
dave
diane
no
```

Success

The *true/0* or the *otherwise/0* predicates are identical apart from their names and they always succeed when they are called. They effectively act as "No operations" procedures.

Repeating Sequences of Clauses

You can repeat a sequence of clauses using the 'Edinburgh' *repeat/0* predicate in conjunction with *fail/0*. *repeat/0* always succeeds even on backtracking so when a failure occurs after a repeat the program cannot backtrack past the call to repeat.

Repeat-fail loops are standard practice in Prolog due to their efficiency; this is due to the fact that any environment bindings are undone each time round the loop. For example, consider the following program for reading terms in from a source file and writing them to the screen.

```
listfile(File) :-
        see(File),
        repeat,
        read_write.
```

```
read_write:-
        eof,
        !,
        seen .

read_write:-
        eread(Term,Vars),
        ewrite(Term,Vars),
        write('.'),
        nl,
        fail.
```

The variables *Term* and *Vars* are unbound each time round the loop and therefore do not take up any space on the stack.

Unique to ***WIN-PROLOG*** is a parameterised version, *repeat/1*, so that you can repeat a sequence of commands a specified number of times. For example if you wanted to print the name 'Martha' ten times on different lines you could do this with the following program:

```
write_martha_ten :-
        repeat(10),
        write('Martha'),
        nl,
        fail.
```

Aborting Programs

Sometimes you will need to terminate a program immediately and return control to the Prolog environment. To do this you can use the *abort/0* predicate. This predicate is most useful in a user-defined error handler when an error has occurred that is serious enough to merit termination of the evaluation.

Suspending Programs

To temporarily suspend a ***WIN-PROLOG*** program the *break/0* predicate is provided. Associated with the *break/0* predicate is the notion of a 'break level' - each time the predicate is called the 'break level' is incremented and each time a suspended execution is resumed the 'break level' is decremented. To return from a break level to a suspended execution, type an end of file character <ctrl-z> or the atom end_of_file at the command line.

Terminating Prolog

WIN-PROLOG can be terminated programmatically using the commands *halt/[0,1]* or *exit/1*. The parameterised *halt/1* can be used to return an exit status value to the underlying operating system.

Debugging

Debugging is a vital part of any programming language. ***WIN-PROLOG*** provides a number of debuggers that enable you to investigate your programs in different ways. In addition to the debuggers there are predicates that provide additional information that may be useful for debugging purposes, these include: checking the style of your code and timing predicates.

The main debugger in ***WIN-PROLOG*** works at the source-level; it allows you to guide the control flow of your program and step through a representation of its source code as it proceeds towards a solution. There are a number of predicates that allow you to specify the interaction with the debugger. The source-level debugger is documented in more detail elsewhere. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Debugging

debug/0	*set the debug mode to on*
'?DEBUG?'/1	*user-defined Prolog program which intercepts calls to the debugger*
debug_hook/1	*system handler for the debug hook*
debugging/0	*write the current status of the debugger to the standard output stream*
force/1	*call a Prolog goal and suspend the debugger for that call*
halt/1	*terminate the current Prolog session and return an error code*
leash/2	*set the interaction with the debugger*
leashed/2	*test or get the leashes on the debugging ports*
ms/2	*time a given Prolog goal*
no_style_check/1	*turn off the specified style of compile-time style checking*
nodebug/0	*switch the debug mode to off*

nospy/1	*remove the spy points from the specified predicates*
nospyall/0	*remove all spy points*
notrace/0	*turn the debug mode to off*
spy/1	*set a spy point on the specified predicates*
style_check/1	*turn on the specified type of compile-time style checking.*
trace/0	*switch the trace mode to on*

Setting the Current Debugger

The current debugger is set by the value of the Prolog flag 'debug_file'. This can be changed using the predicate *prolog_flag/3*

Tracing and Debugging Programs

The currently set debugger may be invoked in two different ways. The first method using the predicate *trace/0* will invoke the debugger at the top-level of the next goal that is run. The second method uses the predicate *debug/0* and will invoke the debugger whenever a spypoint is encountered during the running of a program. The trace and debug modes can be turned off using the predicates *nodebug/0* and *notrace/0*. The current debugging status may be retrieved using the predicate *debugging/0.*

Setting Spypoints

Spypoints are set using the predicate *spy/1* and may be removed using the predicates *nospy/1* and *nospyall/0*.

Setting and Checking the Interaction with the Debugger

The interaction with the debugger can be set using the predicate *leash/1* and the interaction for each port can be checked using the predicate *leashed/1*.

Program Style Checking

When a source code file is loaded into ***WIN-PROLOG*** you have the option of using a built-in style checker to check different aspects of your Prolog programming style. The style aspects that are checked are: the use of isolated variables, the discontiguous definition of a predicate in a file and the re-definition of a predicate in subsequently loaded files.

If you are told that a particular variable is only used once in a clause it may be the case that the variable name has been misspelt and was intended to match with other variables in the clause. In any case it is recommended that variables that are not intended for re-use should be indicated by an underscore character to emphasize the fact.

If you are told that a predicate is defined in the file in a discontiguous manner this may point out that you are not taking the later clauses into account when considering the effect on the program of backtracking. It is best to keep the definitions of predicates in a contiguous block and not interspersed with clauses from another predicate.

If you are told that a predicate is defined in more than one file it may be that you are unaware of the second definition which overwrites the first. If you want the definition for a predicate to be contained in several files you should declare the predicate as multifile using *multifile/1*.

In themselves, the style warnings do not indicate errors but may point to potential bugs. The predicates used to turn style checking on or off are: *no_style_check*/1 and *style_check*/1.

Timing Programs

You can time programs to estimate their efficiency using the predicate *ms/2*:

```
?- ms( member(X,[1,2,3]), T ).        <enter>
X = 1 ,
T = 0 ;                               <space>

X = 2 ,
T = 721 ;                             <space>

X = 3 ,
T = 1282 ;                            <space>

no
```

Definite Clause Grammar

WIN-PROLOG has some built-in facilities provided for implementing language parsers. Here we describe the Definite Clause Grammar (DCG) notation which provides a mechanism for defining the grammar rules of a language. These rules are automatically translated to a Prolog program which defines a parser for the language being defined. For a further detailed discussion of grammars and parsers, please see the chapter '*Using Grammar Rules*' *in* the book '*Programming in Prolog*' by W.F.Clocksin and C.S.Mellish.

Predicates Related to Definite Clause Grammar

'C'/3	*used in the expansion of grammar rules*
expand_dcg/2	*convert grammar rules to Prolog directly*
expand_term/2	*convert between a grammar rule and its Prolog equivalent*
phrase/2	*checks if a sequence of symbols can be parsed as a given type*
phrase/3	*checks if a sequence of symbols can be parsed as a given type*
term_expansion/2	*user-defined hook for grammar rule translation*

Grammars

A grammar of a language is a set of rules that describe what sequences of symbols or words make up valid sentences of that language. A grammar can be used to describe a natural language such as English, or to define a programming language.

For example, the following rules can be thought of as giving a grammar for very simple English sentences.

- A *sentence* can be a *noun phrase* followed by a *verb phrase*.
- A *noun phrase* can be a *determiner* followed by a *noun*.
- A *verb phrase* can be just a *verb*.
- A *verb phrase* can also be a *verb* followed by a *noun phrase*.
- The word 'the' is a *determiner*.
- The words 'boy' and 'house' are *nouns*.
- The word 'likes' is a *verb*.

According to this grammar, the following sequences of words are valid sentences:

the boy likes the house.

the house likes the boy.

The following sequence of words are not valid sentences according to the above rules:

likes the boy.

boy likes boy.

Parsing and Parse Trees

Parsing is the analysis of a sequence of symbols to see if the sequence represents a sentence according to a grammar.

A successful parse of a sequence of symbols will establish the structure of the sentence in terms of the grammar. It will show the way that symbols link together into phrases, the phrases form larger phrases, and ultimately how the phrases link together to form a sentence.

A parse tree is a representation of the structure of a sentence. For example, the following parse tree represents the structure of the sentence:

"the boy likes the house"

according to the informal grammar given in the previous section:

```
sentence(
      noun_phrase(
            determiner(the),
            noun(boy)
           ),
      verb_phrase(
            verb(likes),
            noun_phrase(
                  determiner(the),
                  noun(house)
                 )
           )
     )
```

Grammar Notations

There are many different notations for defining a set of grammar rules. One way is to describe the rules informally in English as we did above. However such a description can lead to ambiguities for more complex grammars. A more formal notation for defining a grammar is the Backus-Naur Form (or BNF). BNF is frequently used to define the grammar of programming languages. The following BNF grammar describes the grammar of simple English sentences introduced above.

```
<sentence>          ::= <noun phrase> <verb phrase>
<noun phrase>       ::= <determiner> <noun>
<verb phrase>       ::= <verb> | <verb> <noun phrase>
<determiner>        ::= the
<noun>              ::= boy | house
<verb>              ::= likes
```

The vertical bar | denotes an alternative. For example, a noun can be either boy or house.

Symbols that appear in sentences of the language are called terminal symbols. The terminal symbols in the above grammar are:

```
the boy house likes
```

Symbols inside angle brackets (e.g. <sentence>) are non-terminal symbols. Non-terminal symbols denote phrases that are constructed from a sequence of one or more terminal symbols.

DCG Notation

The definite clause grammar (DCG) notation is another formalism for defining a grammar. Using this notation a grammar is represented as a set of logical rules. The syntax of these rules is merely a shorthand for ordinary Prolog syntax. When the DCG rules are consulted (or optimized) they are translated into Prolog clauses. (Details of the Prolog representation are described below in the section entitled "The Prolog Representation of the Grammar Rules.")

A grammar specified using the DCG notation can be "executed" as a Prolog program in order to see if a list of symbols represents a valid sentence of the language being defined. Thus a DCG grammar specification not only defines a language; it also defines a parser for the language. In addition, some DCG specifications may be used "in reverse" to generate valid sentences according to the grammar.

Using the DCG notation, a grammar rule has the general form:

head - -> *body*.

which can be read as "*head* takes the form *body*." For example, the grammar rule:

```
sentence - -> noun_phrase, verb_phrase.
```

means "a sentence takes the form of a noun_phrase followed by a verb_phrase."

Using the DCG notation, a non-terminal symbol is represented by a Prolog atom or a structure that is not a list. A terminal symbol is represented by any Prolog term. A sequence of one or more terminal symbols must be written inside a Prolog list. An empty sequence is denoted by the empty list []. If the terminal symbols are character codes they can be written as a string (which is automatically converted to a list).

The left hand side of a grammar rule must be a non-terminal symbol. This non-terminal may be followed by a sequence of terminal symbols (written inside a Prolog list). For further details please see the section below entitled "Terminal Symbols on the Left Hand Side of a Rule".

The right hand side of the rule may contain terminal and non-terminal symbols.

Items on the right hand side may be separated by the Prolog conjunction operator ',' which is read as "followed by."

Items may also be separated by the disjunctive operator ';' which is used to denote alternatives. For example, the rule:

```
noun - -> [boy] ; [house]
```

states that a noun is either “boy” or “house”. (Alternatives may also be defined by giving alternative grammar rules for a non-terminal.)

The right hand side of a grammar rule may contain extra tests that are written as Prolog procedure calls. These tests must be enclosed in curly brackets ('{' and '}'). The

cut symbol may appear on the right side of a rule. It does not need to be enclosed in curly brackets.

For further details see the section below entitled "Adding Extra Tests to DCG Rules."

A Simple Example

This example is a DCG representation of the grammar of simple English sentences.

```
sentence      - - >   noun_phrase, verb_phrase.
noun_phrase   - - >   determiner, noun.
verb_phrase   - - >   verb ; verb, noun_phrase.
determiner    - - >   [the].
noun          - - >   [boy] ; [house].
verb          - - >   [likes].
```

You can use this grammar to parse sentences using the built-in predicates *phrase/2* and *phrase/3*.

```
?- phrase(sentence,[the,boy,likes,the,house]).
yes

?- phrase(sentence,[the,boy,likes]).
yes

?- phrase(sentence,[the,girl,likes,the,house]).
no              % fails because 'girl' is not a known noun

?- phrase(sentence,[the,boy,likes], Rest).
Rest = []
```

Note that *phrase/2* and *phrase/3* may also be used to analyse sub-phrases.

```
?- phrase(verb_phrase,[likes]).
yes

?- phrase(verb_phrase,[likes,the,boy]).
yes

?- phrase(verb_phrase,[the,boy]).
no              % fails because 'the boy' is a noun phrase

?- phrase(noun_phrase,[the,boy,likes,the,house],Rest).
Rest = [likes,the,house]
```

This grammar may also be used to generate sentences and phrases (because the right hand sides of the grammar rules do not contain procedure calls).

```
?- phrase(sentence, S).
S = [the,boy,likes]
S = [the,boy,likes,the,boy]
S = [the,boy,likes,the,house]
S = [the,house,likes]
S = [the,house,likes,the,boy]
S = [the,house,likes,the,house]

?- phrase(verb_phrase, V).
V = [likes]
V = [likes,the,boy]
V = [likes,the,house]
```

Adding Extra Arguments to DCG Rules

The DCG notation has a useful extension which allows non-terminals to have arguments. A non-terminal can have 1 or more arguments.

The first use of this extension is to specify a relationship between the sub-phrases of a sentence. For instance, you could add an extra argument to specify that there must be number agreement between two sub-phrases of a sentence. The following sentences are incorrect because there is no number agreement between the noun phrase and the verb phrase.

```
the boys likes the house
the boy like the house
```

In the first example the noun phrase is plural while the verb phrase is singular, whereas in the second sentence the noun phrase is singular and the verb phrase is plural. The following DCG specification uses a single extra argument which specifies whether each non-terminal (i.e. each phrase type) is singular or plural.

```
sentence(N)          --->   noun_phrase(N), verb_phrase(N).
noun_phrase(N)       --->   determiner(N), noun(N).
verb_phrase(N)       --->   verb(N) ; verb(N), noun_phrase(N).
determiner(_)        --->   [the].
noun(singular)       --->   [boy] ; [house].
noun(plural)         --->   [boys] ; [houses].
verb(singular)       --->   [likes].
verb(plural)         --->   [like].
```

We could "execute" this grammar with the following queries.

```
?- phrase(sentence(X), [the,boy,likes,the,houses]).
X = singular

?- phrase(sentence(X), [the,boys,like,the,houses]).
X = plural

?- phrase(sentence(plural),     [the,boys,like,the,house]).
yes

%
% the following query will fail because the noun
% phrase does not agree with the verb phrase
%
?- phrase(sentence(X), [the,boy,like,the,house]).
no

?- phrase(sentence(plural),     [the,boy,likes,the,house]).
no                          % the sentence is not plural
```

The second use of the extra argument facility is to add some semantic information to the grammar specification. For example, the following rule is part of the definition of the syntax of an arithmetic expression. The extra argument X denotes the value represented by an expression.

```
expression(X) - -> term(Y), "+", expression(Z),
            {X is Y + Z}.
```

(For the complete definition of the syntax of expressions see the example below under "A More Complex Example.")

The third reason for adding extra arguments is to return a parse tree that shows how a sentence was parsed. To generate this structure we must add a single extra argument to each non-terminal symbol. This extra argument will be the parse tree for that non-terminal.

The following grammar of simple English sentences will produce a parse tree to show how a sentence has been parsed.

```
sentence(sentence(N,V))              - ->   noun_phrase(N), verb_phrase(V).
noun_phrase(noun_phrase(D,N))        - ->   determiner(D), noun(N).
verb_phrase(verb_phrase(V,N))        - ->   verb(V), noun_phrase(N).
determiner(determiner(the))          - ->   [the].
noun(noun(boy))                      - ->   [boy].
noun(noun(house))                    - ->   [house].
verb(verb(likes))                    - ->   [likes].
```

The query:

```
phrase(sentence(X), [the,boy,likes,the,house]).
```

will bind X to the following parse tree.

```
sentence(
      noun_phrase(
              determiner(the),
              noun(boy)
             ),
      verb_phrase(
              verb(likes),
              noun_phrase(
                      determiner(the),
                      noun(house)
                     )
             )
     )
```

Adding Extra Tests to DCG Rules

As mentioned above, the right hand side of a grammar rule can contain calls to Prolog procedures. The calls must be enclosed in curly brackets ('{' and '}') to distinguish them from terminal and non-terminal symbols in the rule.

One reason why it is useful to add procedure calls to the body of a rule is to allow extra processing of terminal symbols. For example, consider specifying the syntax of a numeric digit. We could specify a digit as follows:

```
digit --> "0".
digit --> "1".
digit --> "2".
digit --> "3".
digit --> "4".
digit --> "5".
digit --> "6".
digit --> "7".
digit --> "8".
digit --> "9".
```

A more economical definition of a digit would use calls to Prolog procedures:

```
digit --> [N], {N >= "0", N =< "9"}.
```

If the body of a grammar rule contains a call to an output predicate, the output will take place when that rule is selected and parsing reaches that part of the rule.

For example, the grammar rule:

```
sentence -->
        noun_phrase,
        {(write('noun phrase ok'),nl)},
        verb_phrase.
```

will cause the message:

```
noun phrase ok
```

to be printed after the noun phrase has been successfully parsed, but before the verb phrase is parsed.

Calls to cut (!) can appear in the body of a grammar rule, but they must not be enclosed in curly brackets. The use of a cut in a grammar rule commits the parser to using that rule. If the parse of a phrase fails using that rule, it will not attempt to use an alternative rule to parse the phrase.

A More Complex Example

The following grammar illustrates the use of an extra argument in non-terminal symbols. It also illustrates the use of procedure calls in the body of grammar rules. The grammar defines simple arithmetic expressions that are made up of digits and operators.

```
exp(Z)      - - >   term(X), "+", exp(Y), {Z is X + Y}.
exp(Z)      - - >   term(X), "-", exp(Y), {Z is X - Y}.
exp(Z)      - - >   term(Z).
term(Z)     - - >   mynumb(X), "*", term(Y), { Z is X * Y}.
term(Z)     - - >   mynumb(X), "/", term(Y), { Z is X / Y}.
term(Z)     - - >   mynumb(Z).
mynumb(C)   - - >   "+", mynumb(C).
mynumb(C)   - - >   "-", mynumb(X), {C is -X}.
mynumb(X)   - - >   [C], { "0" =< C, C =< "9", X is C - "0"}.
```

The grammar can be used to parse *and* evaluate expressions:

```
phrase(exp(N), "1+2*5-3").
N = 8

phrase(exp(N), "-1+2*3", Rest).
N = 5
Rest = [] ;

N = 1
Rest = "*3" ;

N = -1
Rest = "+2*3" ;

no
```

This grammar cannot be used to generate expressions from known values.

The Prolog Representation of the Grammar Rules

In this section we describe the way in which a grammar rule is translated into an ordinary Prolog clause. (Note that when you consult a DCG specification and then list it, you will see the Prolog clause, not the original grammar rule.)

Each grammar rule is translated into a single Prolog clause that takes an input list and returns an output list. Each clause will parse the input list to see if the initial portion of that list represents a particular type of sub-phrase. If the parse is successful, the clause returns an output list that is the remaining portion (possibly enlarged) still to be analysed. The arguments that represent the input and output lists are not written in the original grammar rule, but are added when the rule is translated. For example, the grammar rule:

```
verb_phrase --> verb.
```

will be automatically translated into an ordinary Prolog clause of the form:

```
verb_phrase(In, Out) :- verb(In, Out).
```

If this clause is invoked to parse the input list:

```
[likes,the,boy]
```

then the output list will be bound to:

```
[the,boy]
```

An input and output argument will be added to each non-terminal on the right hand side of the rule.

For example the grammar rule:

```
sentence --> noun_phrase, verb_phrase.
```

will be translated into the Prolog clause:

```
sentence(S, S0) :-
        verb_phrase(S, S1),
        noun_phrase(S1, S0).
```

The logical reading of this rule is:

> the difference between *S* and its tail end sub-list *S0* is a sentence if the difference between *S* and its tail end sub-list *S1* is a verb phrase and the difference between *S1* and its tail end sub-list *S0* is a noun phrase.

Terminal symbols in the grammar rule are translated into calls of the form:

'C'(*S1*,*X*,*S2*).

where *S1* is the input list whose head is the terminal *X*. *S2* is the remaining portion of the list still to be analysed by the parser.

The definition of 'C'/3 is:

```
'C'([X|S],X,S).
```

For example, the grammar rule:

```
determiner --> [the].
```

will be translated into:

```
determiner(S0, S1) :-
        'C'(S0, the, S1).
```

Procedure calls in the body of a grammar rule are translated literally. For example, the rule:

```
exp(X) --> term(Y), "+", exp(Z), {X is Y + Z}.
```

is translated into:

```
exp(X, S0, S1) :-
        term(Y, S0, S2),
        'C'(S2, 43, S3), % 43 is the character code for '+'
        exp(Z, S3, S1),
        X is Y + Z.
```

Notice that the input and output lists inserted by the grammar rule translator are placed *after* any arguments given in the original grammar rules.

In summary, we have seen that the grammar rule translator uses a pair of lists to represent a single phrase. The phrase is defined to be the difference between the input list and the output list. For example, the following phrase (of type "determiner"):

```
[the]
```

would be represented by the list pair:

```
input list  = [the,boy,likes]
output list = [boy,likes]
```

This list pair is known as a difference list.

Terminal Symbols on the Left-Hand Side of a Rule

The left hand side of a grammar rule may contain terminal symbols. These terminal symbols must appear after the non-terminal symbol, and they must be written as a Prolog list. For example, the following rule is valid:

```
non_terminal, [t1, t2] - -> [t3].
```

Terminal symbols on the left hand side of a rule are inserted into the input sequence being parsed, replacing the corresponding terminal symbols that appear on the right hand side of the rule. For example, if the above rule is used to parse the input sequence:

```
[t3,t4,t5]
```

then the output sequence generated by this rule will be:

```
[t1,t2,t4,t5]
```

This facility allows a given input sequence to be modified by the parser so that it conforms to a standard form that can then be parsed by other grammar rules.

For example, we might want to analyse the command:

```
eat your cabbage
```

in the same way as we would analyse the standard form of such a command :

```
you eat your cabbage
```

By placing non-terminals on the left hand side, we can translate the first version of the command to the second, more correct version:

```
sentence              - -> imperative, verb_phrase, noun_phrase.
imperative, [you]     - -> [].
imperative            - -> [].
verb_phrase           - -> pronoun, verb.
noun_phrase           - -> poss_pronoun, noun.
pronoun               - -> [you].
verb                  - -> [eat].
poss_pronoun          - -> [your].
noun                  - -> [cabbage].
```

The imperative non-terminal will be translated to the following Prolog clauses:

```
imperative(_A, _B) :-
        'C'(_B, you, _A).
imperative(_B, _A).
```

Dictionaries

WIN-PROLOG has three built-in dictionaries that tell you the currently defined atoms, the currently open files and the currently defined predicates. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Dictionaries

dict/2	*return a dictionary of atoms*
fdict/2	*return a dictionary of files*
pdict/4	*return a list of predicates matching the given type and mask*
wdict/2	*return the currently open windows*
wfdict/2	*return the open fonts*
wmdict/2	*return the defined menus*

The Atom Dictionary

The atoms that are currently defined can be found in the atom dictionary. This dictionary can be found using the predicate *dict/2*.

The File Dictionary

To help with the maintenance of currently open files *WIN*-PROLOG has a built-in file dictionary. The file dictionary can be accessed directly,using the predicate *fdict/2*, so as to allow you to control the maintenance of files in your own manner.

We recommend that you do not mix indiscriminately between logical file handling predicates and lower-level ones. For example, the predicates *fcreate/5* and *fclose/1* operate directly on the file dictionary, so whichever filename you use with these predicates is either added or removed from the file list. The predicates *open/[2,3]* and *close/1* make use of the logical file names, so some processing is done to the filename to convert it into its absolute form before adding it to the file dictionary. Let's assume you are in the *WIN*-PROLOG directory, it would not be a good idea to make both of the following calls to *open/2* and *fcreate/5*:

```
?- open( 'examples\meals.pl', read ).

?- fcreate( meals, 'examples\meals.pl', 0, 0, 0 ).
```

If you did make these calls and then examined the file dictionary with the following call to *fdict/2*:

```
?- fdict( 0, Dict ).
Dict = ['C:\Program Files\WIN-PROLOG 4100\EXAMPLES\MEALS.PL',meals]
```

you will see two different results of trying to open this file. This duplicate access of the same file can lead to unpredictable behaviour.

The Predicate Dictionary

The predicates currently loaded into the system can be obtained from the predicate dictionary. They are split into two major categories: system-defined predicates and user-defined predicates, these categories are then sub-divided into four groups: compiled, optimized, assembler and external predicates. Compiled predicates have been loaded into the system from source code. Optimized predicates have been loaded from object code files. Assembler predicates are defined in a dynamically linkable assembler code module.

Currently defined predicates and their types can be found using *pdict/4*.

DOS Handling

To enable you to run external programs or initiate a DOS shell, ***WIN-PROLOG*** provides several predicates for interfacing with DOS. You can also retrieve values from the ***WIN-PROLOG*** command line switches and return information on the current version. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to DOS Handling

dos/0	*initiate a DOS shell*
dos/1	*initiate a DOS shell and run the given command*
exec/3	*execute an external program*
switch/2	*set or get the value of a **WIN-PROLOG** command line switch*
ver/4	*return information on the current version of **WIN-PROLOG***

Running a DOS shell

You can initiate a DOS shell from within ***WIN-PROLOG*** using either of the predicates *dos/[0,1]*. *dos/0* will simply initiate a shell, while *dos/1* will initiate a shell with the given command. For example: the following command will initiate a DOS shell and run the DOS command DATE (to allow you to either see or set the system date):

```
?- dos(date).
```

To escape from the DOS shell back into ***WIN-PROLOG*** type EXIT at the DOS command line.

Running a Command

You can also run a DOS command from within ***WIN-PROLOG*** without having to initiate a DOS shell by using the predicate *exec/3*.

The *exec/3* predicate behaves differently in the two systems: ***DOS-PROLOG*** and ***WIN-PROLOG***. In ***DOS-PROLOG*** when the *exec/3* goal is run, control is given exclusively to the DOS command specified in the call to *exec/3*, any further activity pending in ***DOS-PROLOG*** is suspended until the command relinquishes control. In ***WIN-***

PROLOG, because Windows is a multi-tasking environment, when the *exec/3* goal is run control is returned to ***WIN-PROLOG*** immediately. For example, consider the following goal:

```
?- exec('pro386.exe','',X), beep(100,100).
```

Under DOS, the above goal will run a new incarnation of ***DOS-PROLOG*** (assuming enough space is available) and only after this application has been quit will the original ***DOS-PROLOG*** go on to the *beep/2* call.

```
?- exec('C:\Program Files\WIN-PROLOG 4100\pro386w.exe','',X), beep(100,100).
```

Under Windows, the above goal will run a new incarnation of ***WIN-PROLOG*** and then immediately go on to the *beep/2* call.

Retrieving Command-Line Switches

WIN-PROLOG has 26 built-in named storage locations, their values can be retrieved using the predicate *switch/2*. Initially they are used to store any switches that were specified on the command line when ***WIN-PROLOG*** was invoked. Once ***WIN-PROLOG*** has been loaded the storage locations may be used freely.

Getting Information about *WIN-PROLOG*

Information on the currently running ***WIN-PROLOG*** system can be found using *ver/4*. The information returned includes: the system name, the version number, the date of creation and the serial number.

Error Handling

There are a number of predicates in *WIN-PROLOG* that are associated with error handling. These allow you to abort an evaluation, flush the current input stream, define your own error handler, define the handling of unknown predicates and return an error message for a given number. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Error Handling

'?BREAK?'/1	*user-defined hook to gain control after a break interrupt*
'?ERROR?'/2	*user-defined error handler*
abort/0	*abort the current program*
break/0	*suspend the current execution and start supervisor*
break_hook/1	*default break hook in development environment*
catch/2	*catch the error code generated by a given goal*
catch/3	*same as catch/2 but also return the predicate that generated the error*
error_hook/2	*system defined behaviour for error handling*
error_message/2	*return an error message for an error number*
flush/0	*flush the current input stream*
throw/2	*throw a numbered error*
unknown_predicate_handler/2	*user-defined fact that defines the handling of unknown predicates*

Defining Your Own Error Handler

You can create your own error handler by making a definition for *'?ERROR?'/2* this can be used in conjunction with the *abort/0* and *flush/0* predicates below to ensure the smooth handling of errors.

Aborting the Current Evaluation

In an error handler, when a serious error occurs, it is usually necessary to terminate the evaluation that was taking place. The predicate *abort/0* can be used in this way.

Flushing the Input Buffer

If an error occurs while reading from an input buffer it is usually a good idea to flush the input, prior to continuing, to remove any characters remaining in the buffer. The predicate *flush/0* is provided for this purpose.

Defining an Unknown Predicate Handler

If the current evaluation contains a call to an undefined predicate, ***WIN-PROLOG*** will, by default, raise an error. This behaviour can be modified to failure whenever an unknown predicate is called, by changing the value of the 'unknown' flag from 'error' to 'fail' using *prolog_flag/3*. In addition you can create a definition for *unknown_predicate_handler/2* which allows you to specify a goal to be run whenever an unknown predicate is called.

For example, to report every unknown predicate called and cause failure, define the following:

```
?- assert(unknown_predicate_handler(U,unknown(U))).
```

where the definition of *unknown/1* is:

```
unknown(Unknown) :-
        write('predicate not currently defined: '),
        write(Unknown),
        fail .
```

Getting the Error Messages and Their Numbers

When an error occurs it is reported as a number and either *'?ERROR?'/2*, if such a definition exists, or the default error handler is called. The error number reported can be converted into a meaningful message using *error_message/2*. The default error handler, *error_handler/2*, performs this translation to provide you with a meaningful error message as well as dealing with specific types of error.

Catching and Throwing Errors

The predicates *catch/2* and *throw/2* are useful when handling errors within Prolog programs. They can work in a nested fashion where each call to *throw/2* sends an error number back to the last *catch/2* that was put on the call stack. If there is no *catch/2* remaining on the call stack then the error sent by *throw/2* is generated as an actual error.

The arguments of *catch/2* are as follows:

catch(*-Number*,*+Goal*)

where *Number* is a number returned as a result of running the *Goal*. The value of *Number* can be either 0 if the goal succeeds, -1 if the goal fails or an error number sent by a call to *throw/2* executed within the goal.

The predicate *throw/2* has the following arguments

throw(*+Number*,*+Goal*)

where *Number* is the error number returned. For example:

```
throw(123,foo(A,B,C)).
```

generates error number 123 in the goal "foo(A,B,C)".

Error Handling - An Example

An example of a useful error handler could be one that reports all errors with a simple message to the console, before aborting execution. The following definition for *'?ERROR?'/2* will do just this:

```
'?ERROR?'( Number, Goal ) :-
        errmsg( Number, Message ),
        output( 0 ),
        writeq( error(Number) - Message - Goal ),
        nl,
        abort.
```

After compiling this program, the error handler can be tested:

```
?- foo. <enter>
error(20) - 'Predicate Not Defined' - foo

Aborted
```

Files and Directories

It is likely, in an application, that you will need to access both files and directories. ***WIN-PROLOG*** has a number of built-in predicates allowing easy manipulation of the underlying file-directory structure. There are two levels of file handling predicates, some higher-level logical filename predicates and some faster low-level predicates. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Files and Directories

absolute_file_name/2	*converts from a relative to an absolute file specification*
absolute_file_name/3	*convert between a relative and an absolute file specification using options*
attrib/2	*set or get file attributes*
chdir/1	*choose or return a directory*
close/1	*close the named file*
del/1	*delete a file*
dir/3	*get a file directory*
env/1	*get all environment strings*
fclose/1	*close a file*
fcreate/5	*create or open a disk or memory file with the given access mode and character encoding*
fdata/5	*retrieve information about an open disk or memory file*
fdict/2	*return a dictionary of files*
file/3	*return selected information about the named file*
file_search_path/2	*user-defined fact specifying a path name*
fname/4	*convert a file name into parts*

library_directory/1	*defines a library directory*
mkdir/1	*make a directory*
open/2	*open a file with the given access mode*
open/3	*open a file with the given access mode and character encoding*
ren/2	*rename a file*
rmdir/1	*delete a directory*
stamp/1	*get or set a file date and time stamp*

Low-Level Vs Logical Filenames

In ***WIN-PROLOG*** there are two levels of file handling predicates. The 'logical' filename predicates and the low-level machine code predicates that are used to implement them. The main difference between these two sets of predicates is that the logical filename predicates automatically convert all file names into their absolute path name, whereas the low-level predicates can also use relative path names.

Incompatibilities can occur when trying to use a low-level file handling predicate to refer to a file that has been opened by a logical filename predicate (or vice-versa). For example, if you assume that you are in the ***WIN-PROLOG*** home directory: you could load a file using the following logical filename predicate *open/2*:

```
?- open('eg\meals.pl',read).
```

if you then tried to close this file using the low-level file handling predicate *fclose/1*:

```
?- fclose('eg\meals.pl').
```

even though the same file name was used in both cases this call would fail. A quick look at the file dictionary will tell us why:

```
?- fdict( 0, Files ).
Files = ['C:\PRO386\EG\MEALS.PL']
```

the names simply do not match at the system level. If however we run the following goal:

```
?- absolute_file_name('eg\meals.pl',Abs), fclose(Abs).
Abs = 'C:\PRO386\EG\MEALS.PL'
```

the file is successfully closed, because, the file name is converted into its absolute equivalent. For this reason it is recommended that the two sets of predicates are used independently of one another. Using the logical filename predicates when you wish to

make use of logical file names and using the low-level predicates when efficiency is a more important consideration.

Logical File Handling

The File Search Path Mechanism

To aid the accessing of files within programs, ***WIN-PROLOG*** provides a flexible mechanism for the easy maintenance of directory structures. The paths for directories can be abstracted to logical names which can then be used within programs. In this way an application can be moved to a different directory hierarchy or to a completely new file system, with a minimum of effort. This mechanism is similar to the file search path mechanism found in Quintus Prolog. An alias for a directory structure can be defined by asserting a fact of the form:

```
file_search_path( Alias, DirSpec ).
```

Getting Absolute Filenames

Absolute filenames can be obtained from relative or logical filenames using the predicates *absolute_file_name/[2,3]*. For example, if you wanted to find the absolute path name of a file in the EXAMPLES sub-directory of ***WIN-PROLOG*** you could ask either of the following two queries:

```
?- absolute_file_name(examples(meals),Path).
Path = 'C:\PRO386\EXAMPLES\MEALS.PL'

?- absolute_file_name(prolog('examples\meals'),Path).
Path = 'C:\PRO386\EXAMPLES\MEALS.PL'
```

Note that in the absence of an extension a '.PL' extension is assumed.

Opening Files

The predicate *open/2* can be used to open a named file with the given attributes; the predicate *open/3* can be used to open a named file with the given attributes and character encoding. These predicates can make use of the logical filenames, so care must be taken if you plan to maintain the file dictionary directly. Otherwise, use *close/1* to close any files opened with *open/[2,3]*.

Closing Files

Files may be closed using the predicate *close/1*. This predicate can make use of logical filenames and should be used in conjunction with *open/[2,3]*.

Low-level File Handling

If you want faster file access, there are a number of low-level file handling predicates. These predicates cannot make use of the logical file names and are documented in the 'Technical Reference'.

Garbage Collection and Memory

WIN-PROLOG has a number of built-in predicates for determining and maximising the amount of memory available to *WIN-PROLOG* in its six memory areas. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Garbage Collection and Memory

free/9	*return the free space available in WIN-PROLOG's memory areas*
total/9	*return the total space allocated to WIN-PROLOG's memory areas*
garbage_collect/0	*invoke the garbage collector explicitly*
garbage_collect/1	*invoke the garbage collector explicitly for given memory area*
gc/0	*enable the garbage collector*
gc/1	*perform an explicit garbage collection*
nogc/0	*disable the garbage collector*
statistics/0	*display statistics about the current status of the system*
statistics/2	*get individual memory statistics*
stats/4	*get timer and garbage collection statistics*
ver/1	*output the standard banner*
ver/4	*output the standard banner*

Determining Free Memory

The amount of free memory in *WIN-PROLOG* can be found using the predicate *free/9* where each of the variables refers to a memory area. The memory areas are the backtrack stack, the local stack, the reset stack, the term heap, the text heap, the program heap, the system stack, the string input buffer and the string output buffer.

As an example, this predicate can be used to determine whether a goal is deterministic or not:

```
backtrackpoints(Goal) :-
        free(OldBacktrack,_,_,_,_,_,_,_,_),
        Goal,
        free(NewBacktrack,_,_,_,_,_,_,_,_),
        (OldBacktrack =:= NewBacktrack
        ->
                write('No backtrack points'),
                nl
        ;
                write('Backtrack point found'),
                nl) .
```

The "backtrackpoints/1" program works by comparing the size of the backtrack stack before and after the goal has been called and then comparing the two values, if the values are different then there are backtrack points, otherwise, if the values are identical, the program is deterministic.

```
?- backtrackpoints( member(X,[a,b,c]) ).        <enter>
Backtrack point found
X = a ;                                         <space>

Backtrack point found
X = b ;                                         <space>

Backtrack point found
X = c ;                                         <space>

no
```

Garbage Collection

The program and the text heaps can be garbage collected to maximise the amount of contiguous memory available. Normally this process is completely automatic but may be explicitly called using the predicates *garbage_collect/0*, which collects both memory areas, and *garbage_collect/1*, which collects either the term or the text heap. Explicit garbage collection can be turned on or off using the predicates *gc/0* and *nogc/0* (note: these predicates do *not* effect automatic garbage collection).

Getting Program Space Statistics

The status of the ***WIN-PROLOG*** memory areas can be printed to the current output stream using *statistics/0* the individual statistics referenced by name can be returned using *statistics/2*. For example, to see the current backtrack stack value you could use the following call to *statistics/2*.

```
?- statistics(backtrack_stack,BS).
BS = 65356
```

Getting Version statistics

The standard banner, including memory statistics, can be printed to the current output stream using *ver/1*. Individual aspects of the version information, including the system name, the version number, the date of creation and the serial number can be obtained with *ver/4*.

Input and Output

Input and output in ***WIN-PROLOG*** makes use of standard 'Edinburgh' predicates as well as some faster low-level predicates. The topics covered in this chapter are: term I/O, setting I/O streams, temporarily redirecting I/O, file pointer positioning, formatted I/O, character I/O, outputting format characters and copying data from file to file. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Input and Output

Predicates for Setting I/O Streams

input/1	*set input from a file, device or string*
output/1	*set output to a file, device or string*
see/1	*set the current input stream*
seeing/1	*return the current input stream*
seen/0	*reset the current input stream to the standard input stream*
tell/1	*set the current output stream*
telling/1	*return the current output stream*
told/0	*reset the current output stream to the standard output stream*

Predicates for Temporarily Redirecting I/O

<~/2	*re-direct input to a file or a string*
~>/2	*re-direct output to a file or a string*

Predicates for Positioning File Pointers

at_end_of_file/0	*checks to see if the input file pointer is at end of file*
at_end_of_line/0	*test whether end of line has been reached for the current input stream.*
eof/0	*test to see if the input stream is at end of file*

find/3	*find or copy up to a string in the current input stream*
flush/0	*flush the current input stream*
inpos/1	*set the input stream position*
outpos/1	*sets the output stream position*
skip/1	*skip to just after the specified character code value on the current input stream*
skip_layout/0	*skip past the white space characters on the current input stream*
skip_line/0	*skip the remaining input characters of the current line*
skip_term/0	*skip the remaining input characters up to the end of a term*
stream_position/2	*get the current position of the specified stream*
stream_position/3	*get the current position of the specified stream*

Formatted I/O Predicates

fread/4	*formatted read of a term*
fwrite/4	*formatted write of a term*

Character I/O Predicates

display/1	*write a term to the standard output stream in standard prefix notation*
elex/1	*set, reset or get the edinburgh flag*
eprint/1	*print a quoted edinburgh term to the current output stream*
eprint/2	*same as eprint/1 but with the ability to output variable names*
eprint/3	*same as eprint/2 but with added priority*

eread/1	*read an edinburgh term from the current input stream*
eread/2	*same as eread/1 but with an added variable list*
etoks/1	*read an edinbugh token list from the current input stream*
etoks/2	*read an edinbugh token list from the current input stream with variable names*
ewrite/1	*write an unquoted edinburgh term to the current output stream*
ewrite/2	*same as ewrite/1 but with the ability to output variable names*
ewrite/3	*same as ewrite/2 but with added priority*
get/1	*read a non-white-space character from the current input stream*
get0/1	*read a character from the current input stream*
getb/1	*get a byte (8-bit character code) direct from keyboard*
getx/2	*input a byte, word or dword*
op/3	*declare an operator with a given precedence and type*
portray_clause/1	*write a clause to the current output stream in listing format*
print/1	*print a term to the current output stream*
printq/1	*print a quoted term to the current output stream*
prompt/2	*get or set the Prolog prompt*
put/1	*write a character to the current output stream*
putb/1	*character output direct to screen*

putx/2	*output a byte, word or dword to the current output stream*
read/1	*read a term from the current input stream*
sysops/0	*re-install all of the system-declared operators*
skip_term/0	*skip the remaining input characters up to the end of a term*
vars/2	*return a named list of vars in a term*
write/1	*write a term to the current output stream*
write_canonical/1	*write a term to the current output stream in canonical form*
writeq/1	*write a quoted term to the current output stream*

Predicates for Outputting Format Characters

nl/0	*start a new line on the current output stream*
tab/1	*write the given number of spaces to the current output stream*

Predicate for Copying Data From File To File

copy/2	*copy data from the current input stream to the current output stream*
find/3	*find or copy up to a string in the current input stream*

Keyboard and Screen I/O

keys/1	*get the system key status*
grab/1	*check for a byte (8-bit character code) direct from keyboard*
ttyflush/0	*flush the user output stream*

ttyget/1	*read a non-white-space character from the user input stream*
ttyget0/1	*read a character from the user input stream*
ttynl/0	*start a new line on the user output stream*
ttyput/1	*write a character to the user output stream*
ttyskip/1	*skip to just after the specified character code value on the user input stream*
ttytab/1	*write the given number of spaces to the user output stream*

Sound Output

beep/2	*sound a beep of the given duration and frequency*

Standard and Current I/O Streams

All input (whether term, character or formatted) is read from an input stream. An input stream can be the user's terminal or a file. The standard input stream is the user's terminal (i.e. the keyboard buffer) and is called "user".

Similarly all output is written to an output stream. An output stream can be the user's terminal or a file. The standard output stream is the user's terminal (or screen) and is also called 'user'.

At any given time there is a current input stream and a current output stream. Initially these streams refer to the standard input stream and the standard output stream respectively. However, they can be changed using the *see/1* and *tell/1* predicates.

Predicates such as *read/1* and *write/1* make use of the current input and output streams. Predicates such as *ttyget/1* and *ttyput/1* make use of the standard input and output streams.

Setting I/O Streams

WIN-PROLOG allows you to control the current I/O streams using the standard edinburgh predicates: *see/1*, *seeing/1*, *seen/0*, *tell/1*, *telling/1* and *told/0*. A program that uses *see/1*, *tell/1*, *seen/0* and *told/0* to copy all the Prolog terms in one file to another could be:

```
prolog_copy(File1,File2) :-
  see(File1),                    %open the input file
  tell(File2),                   %open the output file
  repeat,                        %start a repeat loop
  read(Term),                    %read a term
  ( Term \= end_of_file          %if the term not end_of_file
  -> writeq(Term),               %then output the term
    write('. '),                 %output a term terminator
    nl,                          %output a new line
    fail                         %fail back to the repeat
  ; seen,                        %else close the input
    told ) .                     %and close the output
```

A useful feature of I/O programming in Prolog is that the I/O streams are set independently from the predicates that actually perform the I/O. This means that the logic of an I/O program can be debugged and tested on the default streams (keyboard and screen) and then when everything works successfully the intended I/O streams can be used. Our prolog_copy/2 program could be re-written without using calls to the stream setting predicates as follows:

```
prolog_copy :-
  repeat,                        %start a repeat loop
  read(Term),                    %read a term
  ( Term \= end_of_file          %if the term not end_of_file
  -> writeq(Term),               %then output the term
    write('. '),                 %output a term terminator
    nl,                          %output a new line
    fail                         %fail back to the repeat
  ; true ) .                     %else suceed don't repeat
```

This program could then be tested on the command line using the default keyboard and screen I/O streams. This is demonstrated in the following example where the user input is shown in bold text. The *prolog_copy/0* predicate will loop continually, asking for input, until the end of file marker is reached. In *LPA-PROLOG* the end of file marker may be either the '~Z' character or the term end_of_file.

```
| ?- prolog_copy.
|: foo.
foo.
|: foo(A).
foo(_0005035E).
|: foo(B) :- bar(B).
foo(_0003C052) :- bar(_0003C052).
|: end_of_file.
yes
```

Having been tested, the *prolog_copy/0* predicate could then be tried in earnest on two files. The following example makes a copy of the Prolog file 'SOURCE.PL' into the file 'COPY.PL':

```
?- see('source.pl'),tell('copy.pl'), prolog_copy, seen, told.
```

In addition to the Edinburgh predicates there are two lower-level stream I/O predicates: *input/1* and *output/1*. When performing file I/O there is no real advantage to using these predicates. They do have a special function with regard to strings and this is covered in the section titled "Random Access String I/O" below.

Temporarily Redirecting I/O

The current I/O streams can be redirected for the duration of a Prolog call using the predicates *~>/2* and *<~/2*. This can be extremely useful in ***WIN-PROLOG*** for Windows where I/O to windows or DLLs must be done using the string data type. As an example of this type of use, consider the following:

```
?-      wtcreate(fred,`Fred`,10,10,200,400),
        write('The normal output'),
        write('The redirected output') ~> S,
        wtext((fred,1),S),
        nl .
```

This program, when run on ***WIN-PROLOG***, will do the following: create a window called 'fred', write the text 'The normal output' to the current output stream, redirect the text 'The redirected output' into a string represented by *S*, output the string *S* to the window called 'fred' and finally write a new-line to the current output stream.

Both the input and output streams can be re-directed at the same time. We can use our re-written *prolog_copy/0* predicate, defined above, to copy from a file into a string using the following query:

```
?- ( prolog_copy ~> Copy ) <~ 'source.pl'.
```

This re-directs the input of the *prolog_copy/0* to be the file 'source.pl' and the output to the string returned in the variable *Copy*.

Positioning File Pointers

The predicates available for file pointer positioning allow you to test the position of the file pointer, find text, skip text and position the pointer explicitly.

Testing Input Boundary Conditions

An important factor in the successful handling of the input data is the ability to detect boundary conditions. The input end of line and end of file boundaries can be tested using the predicates *at_end_of_line/0* and *at_end_of_file/0* respectively. The following example takes each line from the current input stream and writes it to the current output stream with a comment that gives the line's number. A test is made for the end of file and end of line boundaries before each character is input.

```
/* if at the end of file stop */
number_lines(LineNumber) :-
 at_end_of_file.                    %test for end of file

/* if at the end of line output the line count */
number_lines(LineNumber) :-
 at_end_of_line,                    %test for end of line
 write(' % Line number '),          %write a comment
 write(LineNumber),                 %and the line number
 skip_line,                         %skip to the next line on input
 nl,                                %output a new line
 NewLineNumber is LineNumber + 1,   %increment line count
 number_lines(NewLineNumber).       %recurse with new line count

/* if not at end of file or line input and output a character */
number_lines(LineNumber) :-
 get0(X),                           %get a character
 put(X),                            %output a character
 number_lines(LineNumber).          %recurse
```

This program can take either a file or string as input. The following query tests the *number_lines/1* program with a string:

```
?- number_lines(1) <~ `Line One~MLine Two~M`.
Line One % Line number 1
Line Two % Line number 2

yes
```

Finding Text in an Input Stream

Text can be found in the current input stream using the *find/3* predicate. This predicate is very versatile, effectively performing several functions. At its simplest, it can search for a string in a file or another string, or strip unwanted white space prior to a token on input. Thanks to its output modes, it can also be used to copy from current input to current output up to a specified termination string. This is especially useful when writing search/replace algorithms.

```
?- (find(`c`,3,Found) <~ `abCde`) ~> Before.
Found = `C` ,
Before = `ab`
```

The *skip_layout/0* predicate can be useful when you want to read from the current input stream, but skip past any white space characters (those with ASCII code values of less than 33). Searching will stop either when a printable character is found, or when end-of-file is reached: in either case, *skip_layout/0* succeeds.

Two more skip predicates are available: *skip/1* can be used to skip past the next occurrence of a character with the specified character code and *skip_line/0* which skips to the beginning of the next line of input. We used the *skip_line/0* predicate earlier on

in our *number_lines/1* example, defined in a previous section titled "*Testing Input Boundary Conditions*", to ensure that the next character on input was at the start of the new line.

Setting the Stream Pointer Positions

The file pointer position can be set or checked with the following predicates: *inpos/1*, *outpos/1*, *stream_position/2* and *stream_position/3*. The lower-level *inpos/1* and *outpos/1* are particularly useful as they are able to work on strings as well as files. The predicates, *inpos/1* and *outpos/1*, should be used with care on Unicode files as they refer to byte offsets not character offsets.

The following example uses the *inpos/1* predicate to implement a peek character program that gives a preview of the next character in the current input stream without changing the input pointer position.

```
peek_char(Char) :-
  inpos(I),                    %get the current input position
  get0(Char),                  %get the character code of the next char
  inpos(I).                    %reset the previous input position
```

Formatted I/O

The formatted I/O predicates *fread/4* and *fwrite/4* are designed to provide an efficient way of inputting or outputting particular datatypes. They can be used for: inputting whole lines of text, outputting terms with a given field width, truncating terms and changing numbers into different radices.

Character I/O

The character I/O predicates: *get/1*, *get0/1*, *getb/1*, *grab/1*, *put/1* and *putb/1* all perform I/O with 8-bit character codes (byte values); *get/1*, *get0/1*, *put/1* and *putb/1* also perform with 32-bit character codes. The predicates *getx/2* and *putx/2* can be used to input or output bytes, words and dwords.

Outputting Format Characters

New lines and tabs can be output to the current output stream using the predicates *nl/0* and *tab/1* respectively.

Copying Data From File To File

The predicate *copy/2* has been provided to allow the quick transfer of data from stream to stream.

Keyboard Input

The status of the keyboard can be monitored using the predicates *grab/1* and *keys/1*.

Interpreting Control Keys

To find the status of the control keys (<shift>, <ctrl>, <alt>, <scroll lock>, <num lock>, <caps lock> and <ins>) *WIN-PROLOG* provides the *keys/1* predicate. This returns an integer that represents a 16-bit binary number, where each bit that is set represents the active status of one of the control keys.

Sound Output

The *beep/2* predicate allows you to make a sound of a given frequency and duration. This can be useful when giving warnings or for drawing attention to the screen. For example the following program will beep if the 'Score' is greater than 100:

```
test(Score):-
        (       Score > 100
        ->      beep(1000,1000)
        ;       !,
                fail
        ).
```

List Handling

WIN-PROLOG provides some standard predicates for manipulating lists. The functions available are: appending, reversing, removing items, testing for membership and measuring length. For more details on these predicates please refer to the technical reference manual.

Predicates Related to List Handling

append/3	*join or split lists*
length/2	*get the length of a Prolog list*
mem/3	*return the given member of a term*
member/2	*get or check a member of a list*
member/3	*get or check a member of a list and its position in the list*
remove/3	*remove an element from a list*
removeall/3	*remove all occurrences of an item from a list*
reverse/2	*check or get the reverse of a list*
sort/2	*sort a list into ascending order, removing duplicate terms*
sort/3	*sort a list into ascending order using given key path*

Loading and Saving

WIN-PROLOG has a number of built-in predicates available for loading and saving Prolog programs. Programs can be saved in two formats: source and object format. Object format files load much more quickly than source format files. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Loading and Saving

abolish_files/1	*abolish all predicates associated with the given file*
compile/1	*load source files using the optimising compiler*
consult/1	*load source files using the incremental compiler*
ensure_loaded/1	*load the specified Prolog source or object file(s) into memory, ensuring the most recent versions of files are loaded.*
initialization/1	*declare a goal to be run on loading a file*
load_files/1	*load the most recent versions of files*
load_files/2	*load files according to the specified options*
multifile/1	*declare predicates as being defined in more than one file*
prolog_load_context/2	*get or check information about the current load context*
reconsult/1	*reload source files using the incremental compiler*
save_predicates/2	*save the specified predicates to the named object file*
source_file/1	*get or check the name of a currently loaded program file*

source_file/2	*get or check a currently loaded predicate and program file*
source_file/3	*get or check a predicate, clause count and program file*

Loading Source-Code Files and Object-Code Files

Source files contain programs that are in a human readable form. They can be edited using a text editor. A source file can be loaded using the *ensure_loaded/1*, *load_files/[1,2]*, *compile/1* and *reconsult/1* predicates. By default predicates loaded in this way will be static.

Object files contain programs that cannot be read or edited using a text editor. They can contain dynamic and/or optimised code. An object file is loaded using the *load_files/[1,2]* or *ensure_loaded/1* predicates.

Running Goals Upon Loading

The ':-' operator can be used to automatically run a goal during the consultation of a source file. The goal is run as soon as the ':-' 'declaration' is encountered.

If you are intending to chain files together, using declarations that load other files, you should use the predicate *initialization/1* in the declaration. Initialization goals are run after the loading of the file in which they are contained.

For example: to chain the files FRED.PL and WILMA.PL together you could include the declaration:

```
:- initialization ensure_loaded(fred).
```

in WILMA.PL and the declaration:

```
:- initialization ensure_loaded(wilma).
```

in FRED.PL.

Note: declarations are not maintained in optimized object code files (though they *are* maintained in non-optimized object code files) unless they contain the predicate *initialization/1*. So, if you intend to optimise any source files that contain declarations and you want those declarations to be present in the object code files, you should ensure that all the declarations are of the form:

```
:- initialization Goal.
```

Where *Goal* is the goal to be run.

Loading Predicates From a Source File as Dynamic

By default when a source file is loaded, using either *consult/1*, *reconsult/1*, *ensure_loaded/1* or *load_files/1*, the predicates loaded from the files are considered to be static (this means they cannot be asserted to or retracted from). You can specify within a file that a certain predicate is to be dynamic by having a *dynamic/1* declaration before the definition of the predicate. An easy way of loading an entire file as dynamic code is to use the predicate *load_files/2* with the *all_dynamic(true) option* set. For example, to load the file FRED.PL as dynamic code you could use the following:

```
?- load_files(fred,[all_dynamic(true)]).
```

Predicates Defined In More Than One File

Predicates whose definitions are spread across more than one source file should have the declaration *multifile/1* preceeding the definition of the clauses in each file.

If you want to optimize a multifile predicate you should load all the source files in which the predicate is defined and then use *save_predicates/2* to save the predicate to a file and then optimize that file.

Saving Files

To save individual predicates in object code format, you can use the *save_predicates/2* predicate. For example, if you had the predicates: `foo`/0, `foo`/1 and `foo`/3 in the ***WIN-PROLOG*** database you can save just `foo`/0 and `foo`/3 in the file 'MYPREDS.PC' in the following way:

```
?- save_predicates([foo/0, foo/3], 'MYPREDS').
```

If you want to save the entire contents of the current dynamic workspace as source code to a file, for example the file 'DYNSRCE.PL', you can do so by running the following goal:

```
?- listing ~> 'DYNSRCE.PL'.
```

Maintaining Source Files

In ***WIN-PROLOG*** the association is maintained between the Prolog code and the source files from which it originates. This allows you to have several files loaded and maintain their source separately. The relationship between predicates and their files can be found using *source_file/[1,2,3]*. These predicates return: the currently loaded source files, the predicates associated with those files, the number of the clauses for the predicates and the location of the predicates in the files.

Abolishing Files

If a file has been consulted into ***WIN-PROLOG*** all the predicates associated with that file can be abolished using *abolish_files/1*.

Looking at the Program State

To investigate the workspace, i.e. to see what predicates are currently defined, what type of predicates they are, what files are open, what operators are defined, etc... ***WIN-PROLOG*** provides a number of built-in predicates to provide this information. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Looking at the Program State

current_atom/1	*check or get a current atom*
current_predicate/1	*check or get a current predicate*
current_predicate/2	*check or get a current predicate*
current_op/3	*get the name, type and precedence of a currently defined operator*
def/3	*test for a currently defined predicate and return its type*
defs/2	*return all arities for predicate*
pdict/2	*return a dictionary of predicates*
predicate_property/2	*find the association between predicates and properties*

Predicates and Properties

current_predicate/1 returns the name and arity of a predicate. It can be used to backtrack through the current workspace. For example, if you had defined the following predicates:

```
foo(X,Y) :- foo(X), write(Y).

foo(X) :- foo, write(X).

foo :- write(hello).
```

If you run the goal:

```
?- current_predicate(Pred).
```

It would return the following values:

```
Pred = foo/0 ;
Pred = foo/1 ;
Pred = foo/2
```

current_predicate/2 returns the name of a predicate and the most general term corresponding to that name. It can backtrack through the current workspace. For example, using the workspace defined above, the following goal:

```
?- current_predicate(Name, Term).
```

would produce the following results:

```
Name = foo
Term = foo ;
Name = foo
Term = foo(_00001025) ;
Name = foo
Term = foo(_00001027, _00001029)
```

predicate_property/2 can be used to find out the types of a currently defined predicate. For example, using the workspace defined above, the following call:

```
?- predicate_property(foo, Types).
```

will return:

```
Types = compiled
```

predicate_property/2 can also be used to bactrack and find all the predicates that are built-in to ***WIN-PROLOG***. This can be done with the following goal:

```
?- predicate_property(Pred,built_in).
```

Currently Defined Atoms

The predicate *current_atom/1* can be used to backtrack and find all atoms currently defined in ***WIN-PROLOG*** .

Currently Defined Operators

The predicate *current_op/3* can be used to backtrack and find the precedence, type and name of all the operators currently defined in ***WIN-PROLOG*** .

Getting the Type and Arity of a Predicate

The type and arity of a given predicate can be found using the predicate *def/3*

Getting the Arity of Currently Defined Predicates

The arity of a given predicate can be found using the predicate *defs/2*

Meta-Programming

A meta-level program is one whose data is itself a program. It treats the program as a data structure, processes this structure in some way and may output a transformed version of the program. The program that acts as data is known as the object level program. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Meta-Programming

=../2	*defines the relationship between a structure/atom and a list*
(_ -> _ ; _)	"*if:then:else*"
arg/3	*find the nth argument of a term*
call/1	*call a Prolog goal*
call/2	*call a Prolog goal and return the termination port*
force/1	*call a goal for which a spypoint is currently set*
functor/3	*the relationship between a term, its functor name and its arity*
one/1	*one solution meta-call*

The ability to write meta-level programs is one of the strengths of Prolog in general, and ***WIN-PROLOG*** in particular. The following applications are examples of meta-level programs:

- compilers
- interpreters
- tracer programs
- program transformation tools
- Prolog program editors
- expert systems whose knowledge bases are represented as Prolog programs

Prolog is suitable for writing meta-level programs because it uses the same type of data structure to represent both programs and data. For example the clause:

```
likes(Logician, Person) :- likes(Person, logic).
```

is just the compound term:

```
:-(likes(Logician, Person), likes(Person, logic))
```

Individual components of a clause are also represented by Prolog terms.

Given that Prolog programs can be treated as data, Prolog unification can be used to analyse these programs. The programs can also be manipulated using ***WIN-PROLOG***'s built-in predicates. In particular the meta-level predicates are specifically designed to manipulate terms that represent components of Prolog clauses.

The *arg/3* predicate is used to access specific arguments of a compound term. It can be used to access arguments in the conclusion or body of a clause.

The *functor/3* predicate can be used to access the functor and arity of a compound term. It can also be used to construct compound terms with a given functor and arity. It is a useful predicate for analysing the conclusions and conditions of a clause.

The *=../2* predicate converts between compound terms and lists. It allows a conclusion or condition to be converted to a list. This list can then be processed (e.g. by removing or changing certain elements) and the resultant list converted to a new conclusion or condition. For example, this predicate could be used to add extra arguments to the conclusion and conditions of a clause.

Meta-Programming

The typical 'Edinburgh' syntax for meta-programming uses the *call/1* predicate. For example the *\+/1* predicate would be defined in the following way:

```
\+(Goal ) :- call(Goal), !, fail.
\+(Goal ).
```

In ***WIN-PROLOG*** you could also define the *\+/1* predicate as follows:

```
\+(Goal ) :- Goal, !, fail.
\+(Goal ).
```

Sets of Solutions

Sometimes it is useful when a goal has several solutions to collect all (or some) of those solutions as a list; included in *WIN-PROLOG* are a number of predicates that do just that. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Sets of Solutions

^/2	*existential quantifier*
bagof/3	*find all the instances of a term for which a Prolog goal is true*
findall/3	*find all the instances of a term for which a Prolog goal is true*
forall/2	*generate then test solutions for a goal*
setof/3	*find the set of instances of a term for which a Prolog goal is true*
solution/2	*return the nth soution to a specified call*

Sets and Bags

setof/3 can be used to find the set of solutions for some given term or terms in a goal. The solutions to the goal will be sorted and duplicate elements removed.

bagof/3 can be used to find all solutions to a goal for specified bound variables in the goal.

findall/3 can be used to find all solutions to a goal where all the variables in the goal are bound variables.

Using these predicates to find all solutions to a goal is much more efficient than writing a program which uses backtracking to get the alternative solutions and uses the database (via *assert/[1,2]*) to hold the solutions as they are found.

String Handling

One of the more powerful features of ***WIN-PROLOG*** is a string data type, which exists in addition to the traditional Prolog byte list string type (called char list in ***WIN-PROLOG***). Strings in ***WIN-PROLOG*** allow the compact storage of large quantities of text, and also provide an interface to several special I/O functions. For more details on the following predicates please refer to the 'Technical Reference'.

Predicates Related to String Handling

<~/2	*re-direct input to a file or a string*
~>/2	*re-direct output to a file or a string*
cat/3	*atom and string concatenation*
elex/1	*set, reset or get the edinburgh syntax flag*

Atoms and Char lists

Before discussing strings in detail, a few words should be said concerning their relationship to the traditional Prolog text data types, namely atoms and byte lists (called char lists in ***WIN-PROLOG***). Both these types are of course fully integrated into ***WIN-PROLOG***. Atoms are effectively symbols, or names, which are indivisible, and which identify programs, files, windows, database items and so forth. Char lists are simply linked lists in which each cell contains a 32-bit integer in the range 0h-FFFFFFFFh, representing the character code for a character. In a 32-bit Prolog system, this is rather expensive, with each character requiring 10 bytes of storage! Char lists are generally used in programs which manipulate text on a character-by-character level. In source code, atoms are represented using single quotes:

```
'This is an atom'
```

or, where all characters in the atom are of the same lexical type, without quotes of any kind. Char lists are represented in source code using double quotes:

```
"This is a char list"
```

When printed out by a Prolog program, atoms are given single quotes if they contain mixes of lexical types, or if they contain certain reserved characters; char lists are printed as any other list, for example:

```
[84,104,105,115 ... ]
```

Strings

In *WIN-PROLOG*, the string is a data type which falls somewhere between the atom and the char list. It shares the atom's text storage compactness, but is more easily manipulated. Unlike atoms, strings are not stored on the dictionary, and so may be created faster. Furthermore, while atoms in *WIN-PROLOG* are limited in length to 1024 characters, strings may be up to 3 gigabytes in length. Strings are represented in *WIN-PROLOG* using the backwards quote:

```
`This is a string`
```

For reasons of compatibility with existing Prolog programs, this extension to Prolog syntax can be disabled temporarily (see below). When disabled, the string predicates may still be used, but strings will not be recognised while reading terms from an input stream.

Properties of the Text Data Types

The main properties of the three text data types are summarised in *Table 15*. As can be seen, the *WIN-PROLOG* string data type provides Prolog with considerable power and flexibility.

Property	Atoms	Strings	Char lists
Max length	1024 bytes	3 gigabytes	Limited by Heap Size
Dictionary	yes	no	no
Space per character (0h..FDh)	1.3-2 bytes	1.3-2 bytes	10 bytes
Space per character (FEh..FFFFh)	3.9-6 bytes	3.9-6 bytes	10 bytes
Space per character (10000h.. FFFFFFFFh)	6.5-10 bytes	6.5-10 bytes	10 bytes
Main uses	Predicate, file and other names	I/O streams, data transfer, large scale text processing, special predicates	List processing of text

Table 15 - the properties of atoms, char lists and strings

The major advantages of strings over atoms is that their maximum length is considerably greater, and their creation does not involve dictionary look-up. Compared to char lists, strings are 5-7 times more compact, and do not use up valuable heap space. Strings are also fundamental to a number of very powerful, special purpose predicates, as will be discussed below. For more details on the above, please refer to Appendix L of the 'Technical Reference'.

Atom, Char list and String Conversions

It is possible to convert any of the three text data types into any of the other types, using the three predicates shown below:

```
atom_chars(A,C)      % converts atoms <-> char lists

atom_string(A,S)     % converts atoms <-> strings

string_chars(S,C)    % converts strings <-> char lists
```

The only constraints are that when converting either of the other two types into atoms, the maximum length of the char list or string must not exceed 1024 characters.

Strings and Window Handling

Irrespective of whether you are running *WIN*-PROLOG or *DOS*-PROLOG, one of the important functions of strings is in relation to window input and output. The window handling predicates are described in considerable detail in the 'Technical Reference', but a few words deserve to be said in the present context.

Window Handling in *WIN*-PROLOG and *DOS*-PROLOG

You cannot read text out of windows directly, but thanks to the *wedttxt/2* predicate you can obtain the text as a string. This simple but powerful feature allows you to save window contents, reformat them and show them in other windows at a later point. For example the following:

```
?- wedtsel((fred,1),0,80),wedttext((fred,1),Text), assert(foo(Text)).
```

will return, in *Text*, a string consisting of 80 text characters from the top of the edit window (fred,1), and will store it in an assertion for `foo`/1.

The equivalent in *DOS*-PROLOG would be:

```
?- wedtsel(fred,0,100),wedttext(fred,Text), assert(foo(Text)).
```

as edit windows in *DOS*-PROLOG are not named as a conjunction and exist as top-level windows in their own right.

Later, you could display this information at the beginning of the edit window (fred,1), with the call:

```
?- wedtsel((fred,1),0,0), foo(Text), wedttext((fred,1),Text).
```

Again, the equivalent in *DOS-PROLOG* would be:

```
?- wedtsel(fred,0,0), foo(Text), wedttext(fred,Text).
```

Strings are also used in *WIN-PROLOG* for transferring information to and from some of the built-in dialogs and any DLLs that are present.

Strings and Input/Output

Perhaps the most powerful feature of strings is their ability to act as input and output streams. This property allows you to perform complex text formatting, data type conversions, and other manipulations. String I/O is handled through the I/O redirection predicates *~>/2* and *<~/2*. Normally these are used to divert the input to or output from a single call from or to a file (named by an atom or logical file name). To output to a string, you simply give a variable in place of the output file name, for example:

```
?- write(hello(world,123)) ~> String .
```

will result in the variable being bound as follows:

```
String = `hello(world,123))`
```

To input from a string, you simply give a string in place of the input file name, for example:

```
?- read(X) <~ `foo(Bar). `.
```

will result in the variable being bound as follows:

```
X = foo(_).
```

The sorts of things you can do are manifold, and (in the classic cliche), limited only by your imagination. Here are just two more examples, the first of which copies an entire file (less than 64Kb in size) into a string:

```
?- see(fred), copy(65535,_) ~> FileString, seen.
```

FileString = *<all the contents of file fred>*

The second example converts a number into a formatted atom:

```
?-  fwrite(f,6,3,3.14159) ~> String, atom_string(Atom,String).
    String = ` 3.142`
    Atom = ' 3.142'
```

Term Comparison and Sorting

In this chapter we look at the comparison and sorting of terms within ***WIN-PROLOG***. Unlike the arithmetic comparisons (e.g. *</2*, *>=/2*), the comparison and sort predicates can handle all types of terms, not just numbers. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Term Comparison and Sorting

Unify

=/2	*unification between two terms*
\=/2	*tests for non-unification between two terms*

Comparison

==/2	*check that two terms are identical*
\==/2	*check that two terms are not identical*
eqv/2	*check two terms for equivalence*

Ordering

@</2	*test if one term is less than another*
@=/2	*test if one term is the same as another*
@=</2	*test if one term is equal to or less than another*
@>/2	*test if one term is greater than another*
@>=/2	*test if one term is greater than or equal to another*
@\=/2	*test if one term is not the same as another*
cmp/3	*compare two terms*

compare/3	*find the relationship between one term and another*
Length	
len/2	*return length of a term*
Sorting	
keysort/2	*sort a list of key-value pairs into ascending order*
sort/2	*sort a list into ascending order and remove duplicates*
sort/3	*sort a list into ascending order using a key, do not remove duplicates*
Checking	
occurs_chk/2	*occurs check*
subsumes_chk/2	*check that one term subsumes another*

Sorting

The sorting algorithm used in ***WIN-PROLOG*** is a "list-merge" sort, based on the algorithm of this name described by D E Knuth in volume 3 of his book, "The Art of Computer Programming". The list-merge algorithm is ideally suited to implementation in a Prolog system, since it represents data as linked lists. The advantages of the list-merge sort compared with other algorithms are considerable. Firstly, the algorithm is very fast. Secondly, the algorithm is of the order ln2(n). Thirdly, the list-merge sort is unaffected by the original ordering of data: common algorithms like Quicksort work well most of the time, but have worst cases in which they become very inefficient. The list merge sort has no such cases.

Standard Ordering

The comparison used in sorting and elsewhere in ***WIN-PROLOG*** is able to handle all types of Prolog term. A "standard ordering" is used to relate terms of different types, and this ordering is as follows:

variables	are less than:
integers and floats	which are less than:
atoms	which are less than:
strings	which are less than:
lists	which are less than:
compound terms	which are less than:
true conjunctions	which are less than:
true disjunctions	

Within any one simple type, comparisons are based as follows:

variables	address
integers and floats	numerical value
atoms	character code value
strings	character code value

In structured types, comparison is made on the recursively first element in the two terms. If these elements are identical, then the second elements are compared, and so on until a differing pair of terms is found, or one of the structures is found to be shorter than the other. In the last case, the shorter structure is "less" than the longer one.

Sorting on Keys

The *sort/3* predicate in ***WIN-PROLOG*** allows sorting to be carried out not only on the entire terms in a list, but also on specified subterms. A "path", or list of integers, is given which uniquely identifies which subterm is to be used as the sort key. This is explained in more detail in the documentation of the *sort/3* predicate, but it is illustrated in the following example:

Consider the following database for the 'foo/2' relation.

```
foo(a,10).
foo(g,4).
foo(h,3).
foo(j,1).
foo(c,8).
foo(e,6).
foo(b,9).
foo(f,5).
foo(d,7).
foo(i,2).
```

Using *findall/3* and *sort/3* we can return the information contained in this database in two sorted forms. The first will be sorted according to the first arguments of the 'foo/2' relation:

```
?- findall((X-Y),foo(X,Y),L),sort(L,SL,[2]).
X = _ ,
Y = _ ,
L = [a - 10,g - 4,h - 3,j - 1,c - 8,e - 6,b - 9,f - 5,d - 7,i - 2] ,
SL = [a - 10,b - 9,c - 8,d - 7,e - 6,f - 5,g - 4,h - 3,i - 2,j - 1]
```

The second will be sorted according to the second arguments of the 'foo/2' relation:

```
?- findall((X-Y),foo(X,Y),L),sort(L,SL,[3]).
X = _ ,
Y = _ ,
L = [a - 10,g - 4,h - 3,j - 1,c - 8,e - 6,b - 9,f - 5,d - 7,i - 2] ,
SL = [j - 1,i - 2,h - 3,g - 4,f - 5,e - 6,d - 7,c - 8,b - 9,a - 10]
```

Sorting and Duplicate Removal

One unfortunate feature of the Edinburgh Prolog standard is that the *sort/2* predicate removes duplicate entries from the sorted list. While this can be useful under certain circumstances, more often it is a nuisance. Sorting can be used, for example, to build a very efficient algorithm for counting word occurrences in text. By sorting a list of words without duplicate removal, counting the occurrence of each word is done by checking the length of each contiguous sequence in the sorted list. A similar operation on an unsorted list involves searching the entire list for each word being counted. Duplicate removal within the sorting predicates would prevent the use of this kind of algorithm.

In order to conform with the Edinburgh Prolog standard, ***WIN-PROLOG*** implements the *sort/2* predicate correctly - in other words, so that it removes duplicate entries from the list. However, to give greater flexibility in the use of sorting, the *sort/3* predicate (which is unique to ***WIN-PROLOG***, and therefore not in the Edinburgh standard) does *not* remove duplicates, for example:

```
?- sort([the,dog,and,the,cat],X).
X = [and,cat,dog,the]
```

but:

```
?- sort([the,dog,and,the,cat],X,[]).
X = [and,cat,dog,the,the]
```

In fact, *sort/2* is implemented in terms of *sort/3*, roughly as follows:

```
sort(List,Nodups) :-
        sort(List,Sorted),
        remove_duplicates(Sorted,Nodups).
```

Removal of duplicates from a sorted list is, of course, efficient, but even so it adds a small overhead to sorting. For applications where duplicates either do not occur in the data, or do not need to be removed even if they do, use of *sort/3* will give better performance than *sort/2*, albeit at the cost of portability to other Prologs.

Checking

Executing the following command will generate a 'Term too deep' error:

```
?- X = f(X). <enter>
Term too deep
```

The *occurs_chk/2* predicate can be used to check whether a particular variable occurs within a term:

```
?- occurs_chk( f(X), X ). <enter>
X = _
```

Term Conversion

WIN-PROLOG has a number of built-in predicates for converting between terms. These include converting between: atoms, strings, char lists, lists and terms, structures and lists, numbers and char lists and from terms with uninstantiated variables to terms with instantiated variables. For more details on these predicates please refer to the 'Technical Reference'.

String type conversion is particularly useful in ***WIN-PROLOG*** for Windows because the DLL interface relies on the specific use of strings for the data being transferred.

Predicates Related to Term Conversion

=../2	*defines the relationship between a structure/atom and a list*
atom_chars/2	*converts between an atom and a list of characters*
atom_string/2	*convert from an atom to a* ***WIN-PROLOG*** *string*
copy_term/2	*copy a term with new variables*
lwrupr/2	*convert between lower and upper case*
name/2	*convert between an atom or number and a char list*
number_atom/2	*convert between a number and an atom*
number_chars/2	*convert between a number and a list of characters*
number_string/2	*convert between a number and a* ***WIN-PROLOG*** *string*
numbervars/3	*instantiate the variables in a given term*
string_chars/2	*convert from a list of character codes to a* ***WIN-PROLOG*** *string*

Converting Between Atoms and Char lists

The predicate *atom_chars/2* can be used to convert between atoms and a list of character codes that represent the characters in the atom.

Converting Between Atoms and Strings

The predicate *atom_string/2* can be used to convert between the atom and the string data-type.

Converting Between Char lists and Strings

The predicate *string_chars/2* can be used to convert between a list of character codes and a string.

Term Input and Output

WIN-PROLOG has all the standard 'Edinburgh' term input and output predicates, as well as some fast built-in term I/O predicates for specific functions. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Term Input and Output

current_op/3	*get the name, type and precedence of a currently defined operator*
display/1	*write a term to the standard output stream in standard prefix notation*
elex/1	*set, reset or get the edinburgh flag*
eprint/1	*print a quoted edinburgh term to the current output stream*
eprint/2	*same as eprint/1 but with the ability to output variable names*
eprint/3	*same as eprint/2 but with added priority*
eread/1	*read an edinburgh term from the current input stream*
eread/2	*same as eread/1 but with an added variable list*
etoks/1	*read an edinbugh token list from the current input stream*
etoks/2	*read an edinbugh token list from the current input stream with variable names*
ewrite/1	*write an unquoted edinburgh term to the current output stream*
ewrite/2	*same as ewrite/1 but with the ability to output variable names*
ewrite/3	*same as ewrite/2 but with added priority*

op/3	*declare an operator with a given precedence and type*
portray_clause/1	*write a clause to the current output stream in listing format*
print/1	*print a term to the current output stream*
prompt/2	*get or set the Prolog prompt*
read/1	*read a term from the current input stream*
sysops/0	*re-install all of the system-declared operators*
skip_term/0	*skip the remaining input characters up to the end of a term*
vars/2	*return a named list of vars in a term*
write/1	*write a term to the current output stream*
write_canonical/1	*write a term to the current output stream in canonical form*
writeq/1	*write a quoted term to the current output stream*

Maintaining Variable Names During The I/O Of Terms

The predicates *eread/2*, *eprint/[2,3]* and *ewrite/[2,3]* all contain an argument that returns or defines the relationship between the variables in a term and the name of the variable when it is in printed form. For example, to read in a term and then write the term to the screen keeping the variable names, you could run the following sequence of goals:

```
?- eread(Term,Vars), ewrite(Term,Vars),nl.
```

If you then type in:

```
foo(A,B,C).
```

This will produce the following output:

```
foo(A,B,C)
Term = foo(_0004A426,_0004A430,_0004A43A) ,
Vars = [(A,_0004A426),(B,_0004A430),(C,_0004A43A)]
```

Declaring Operators

Operators are declared using the built-in predicate *op/3*. The form of this predicate is:

```
op(+Precedence, +Type, +Name)
```

where *Precedence* is the operator's precedence (an integer in the range 1 to 1200), *Type* defines the operator type and associativity (e.g. fx), and *Name* is the name of the operator (or a list of operator names). If *Precedence* is 0 then the operator declaration for *Name* is cancelled.

Examples

The following examples show how some of the built-in operators are defined.

```
op(200, xfy, ^).
op(500, fx, [+, -]).
```

It is possible to have more than one operator of the same name. For example, the built-in operator '+' is declared as both a prefix and an infix operator.

The built-in predicate *current_op/3* can be used to find out what operators are currently defined. The format of this predicate is:

```
current_op(?Precedence, ?Type, ?Name)
```

This succeeds if there is an operator called *Name* of type *Type* and with a precedence of *Precedence*. It can be used to backtrack through the list of currently defined operators.

Examples

```
current_op(X, Y, Z).
current_op(500, X, Y).
```

Term Type Checking

WIN-PROLOG has a number of predicates available for checking the types of terms. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to Term Type Checking

atom/1	*test for an atom*
atomic/1	*test for an atom or a number*
callable/1	*check to see if a term is an atom or a compound.*
char/1	*check for an integer representing a character code*
chars/1	*check for a list of integers representing character codes*
compound/1	*test for a compound term*
float/1	*test for a floating point number*
ground/1	*check for completely bound terms*
integer/1	*test for an integer*
integer_bound/3	*generate or test a number between lower and upper bounds*
list/1	*test for a list*
lst/1	*test for a list*
nonvar/1	*test for a non-variable*
number/1	*test for a floating point number or integer*
simple/1	*check for an atom, number or variable*
string/1	*test for a string*
type/2	*return type of a term*

unifiable/2	*check that two terms are potentially unifiable*
var/1	*test for an uninstantiated variable*

Type Checking Predicates

The type checking predicates of the form: Type(*Term*), test a single argument for its Type: i.e. whether it is a variable, number, atom, string, list or compound term. There are a number of additions in ***WIN-PROLOG*** to the standard 'Edinburgh' type checking predicates, such as: *string/1*, *chars/1* etc.

Testing For an Integer Between Bounds

You can test for an integer between specified upper and lower bounds using the predicate *integer_bound/3*. If the argument to be tested is a variable this predicate will generate successive integers between the two bounds.

Switching According to The Types Of Terms

The predicate *type/2* will return the type of any given term. This can be used in programs as an efficient way of testing term types and switching to code that is appropriate to that type of term.

The Clause Database

WIN-PROLOG has some standard 'Edinburgh' syntax predicates available for modifying the Prolog database at run time. For more details on these predicates please refer to the 'Technical Reference'.

Predicates Related to The Clause Database

abolish/1	*delete all the predicates specified by the given argument*
abolish/2	*delete all clauses for the given predicate and arity*
abolish_files/1	*abolish all predicates associated with the given file*
assert/1	*add a clause at the end of the clauses associated with its predicate name*
asserta/1	*add a clause at the beginning of the clauses associated with its predicate name*
assert/2	*assert the clause at the given position*
assertz/1	*add a clause at the end of the clauses associated with its predicate name*
clause/2	*get or check the body of a clause given its head*
clauses/2	*return all clauses matching the given head*
clause/3	*get or check the body and position of a clause given its head*
dynamic/1	*define a predicate to be dynamic*
dynamic_call/1	*call a dynamic procedure safely*
functor/3	*the relationship between a term, its functor name and its arity*

listing/0	*list all the dynamic clauses in the workspace to the current output stream*
listing/1	*list the specified dynamic predicates to the current output stream*
retract/1	*delete a clause that matches the given clause*
retractall/1	*delete all clauses that match the given clause head*
retract/2	*retract a clause at a specified position*
volatile/1	*declare that the clauses for a predicate will not be saved in object files*

Compiled, Optimized, Static and Dynamic Predicates

A *WIN-PROLOG* predicate is either optimized or compiled. Optimized code is static, it cannot be listed (this is useful for hiding the code for programs from end users) and runs faster than compiled code. Compiled code may be either dynamic or static (it c*annot* be a mixture of the two), it can be listed but runs slower than optimized code.

Static procedures can be changed only by completely redefining them using c*onsult/1* or c*ompile/1*. The complete definition of a static predicate can be deleted using *abolish/1* or *abolish/2*.

Dynamic procedures can be modified by adding or deleting individual clauses using the assert and retract procedures. At any time the source code definition of the predicate can be retrieved using the *clause/2* predicate.

If a procedure is defined by being consulted, it is static by default. If you need to be able to add, delete, or inspect the individual clauses of such a procedure, you must make the procedure dynamic.

There are two ways to make a procedure dynamic:

- If the procedure is to be optimized or consulted, then it must be declared to be dynamic before it is defined.
- If the procedure is to be created by assertions only, then the first assert operation on the procedure automatically makes it dynamic.

For example, if you do not have a predicate called 'foo' with three arguments in your database the goal:

```
?- foo(A,B,C).
```

will generate an error. If you then enter the following goals:

```
?- dynamic(foo/3), assert(foo(a,b,c)), listing(foo/3).
```

this will generate the following output:

```
foo(a,b,c).
```

If you then abolish `foo/3`:

```
?- abolish(foo/3).
```

then the goal:

```
?- foo(A,B,C).
```

will fail because the predicate 'foo/3' no longer exists in the database, but does not generate an error because 'foo/3' is defined as dynamic.

The status of a predicate can be tested at any time using the built-in predicate *predicate_property/2*.

Types of Compilation

This appendix discusses the differences between the three types of compilation, namely incremental, hashed and optimised compilation in *WIN*-PROLOG, and indicates why and when each of these types of compilation should be used.

Incremental Compilation: Clause by Clause

The term "incremental compilation" is used to describe a process where each individual program clause is compiled independently, without reference to other clauses. In *WIN*-PROLOG, incremental compilation is performed by handwritten 80386 assembler code, and is extremely fast. First argument indexing and unification instructions are highly optimised, even in the incremental compiler, making it ideal for use in small to medium sized data relations (collections of facts), whether or not these need to be modified dynamically at run time.

Program relations (collections of rules, with or without recursion) are less efficiently compiled, but still run considerably faster than would be possible using an interpreter.

A special feature of incrementally compiled programs is that they can be incrementally decompiled, back into their original source form; this allows full support not only of assert/1 (compilation), but also clause/2, retract/1, and so forth. Apart from this flexible run-time support, incremental compilation also fully supports the *WIN*-PROLOG debuggers.

Hashed Compilation: Instant Access

The term "hashed compilation" is a slight misnomer, as hashing is actually a reversible post-process applied to incrementally compiled programs. Introduced in version 4.200 of *WIN*-PROLOG, this feature allows the creation of a highly optimised hash table for any incrementally compiled predicate that has no variable or variable-headed structure cases in its first argument. The benefit of hashing on small to medium sized relations is relatively slight, but it really comes into its own on large relations containing thousands, or even millions, of clauses.

When an incrementally compiled predicate is hashed, each of its unique first arguments is counted and built into a table with a user-defined amount of headroom, specified by a "fudge factor", or percentage excess. The greater the fudge factor, the fewer the number of mishits are likely to occur during hashing, resulting in a play-off between efficiency and memory requirements. A default setting of 100% gives excellent performance in most circumstances. Because hashing is reversible, it is possible to try different settings on any given relation quickly to find out an optimal fudge factor. Although a hashed relation cannot be modified with assert/1 or retract/1, it is still fully accessible through clause/2, and so forth; furthermore, because hashing can be removed as easily as it has been added, a relation revert to its incrementally compiled status for any such modifications, before (optionally) rehashing. Hashed

compilation also fully supports the ***WIN-PROLOG*** debuggers.

Optimised Compilation: Relation by Relation

The term "optimised compilation" is used to describe a process where each entire relation (collection of clauses for a given predicate and arity) is compiled into a single piece of code. In ***WIN-PROLOG***, optimised compilation performed by a program is written in Prolog which, although somewhat slower at compiling than the incremental compiler, carries out sophisticated analysis on data and control flow within the program, eliminating redundant data transfers and optimising data traffic in general. The optimising compiler can perform multiple argument indexing, and special high speed jump instructions can convert tail recursion into a conventional program loop. Data relations do not normally benefit from being optimised, unless multiple argument indexing is used, but program relations are invariably faster and more space efficient.

Because the optimised compiler converts an entire program relation into a single, monolithic piece of executable code, it is not able to support assert/1, clause/2, retract/1, or the ***WIN-PROLOG*** debuggers. On the other hand, since optimised code cannot be decompiled, it provides a level of security for program source code.

First Argument Indexing

The argument indexing used in ***WIN-PROLOG*** is fairly extensive, but there are some minor differences between the incremental, hashed and optimised compilers in this respect. Firstly, the incremental compiler handles a few more indexing cases than the hashed compiler, which in turn handles a few more than the optimising compiler; secondly, on the other hand, the optimising compiler can index on multiple arguments, which the incremental and hashed ones cannot.

All three compilers index on the type of the first argument; further, if the argument is an atom or an integer, they index on the value. When the first argument is any kind of compound term (tuple, list, conjunction or disjunction), the incremental compiler indexes on the type of its first element, and if this is an atom, on the value. The optimising compiler only performs indexing on the value of an atom in the first element of a compound term, and does not index on the other types when at the head of a term. The following table summarises the indexing capabilities:

Indexing Type	**Inc**	**Hsh**	**Opt**
argument type	*yes*	*yes*	*yes*
argument atom value	*yes*	*yes*	*yes*
argument integer value	*yes*	*yes*	*yes*
tuple head type	*yes*	*yes*	*no*
tuple head atom value	*yes*	*yes*	*yes*

list head type	yes	yes	no
list head atom value	yes	yes	yes
conjunction head type	yes	yes	no
conjunction head atom value	yes	yes	yes
disjunction head type	yes	yes	no
disjunction head atom value	yes	yes	yes
indexing on 1st argument	yes	yes	yes
indexing on other arguments	no	no	yes

The Comparison: Head to Head

The three compilers in ***WIN-PROLOG*** provide complementary, rather than competing services; in an application, it will typically be desirable to use a mix of all types of compiled code. A key consideration is that both incremental and optimising compilers implement indexing through a form of case statement, where successive table entries are checked sequentially, while the hashed compiler implements a true hash table that can give direct access to the clause required. In small or medium relations, the performance improvement is fairly small, and may be offset by other considerations (such as the desire to modify the data relation dynamically, or to disguise the source code by optimising it); in large data relations, the improvement offered by hashing can be very dramatic. The following table summarises the main features and properties of the three compilers:

Feature	**Inc**	**Hsh**	**Opt**
suitable for small data relations	yes	no	yes
suitable for medium data relations	yes	yes	no
suitable for large data relations	no	yes	no
suitable for fast program relations	no	no	yes
supports assert/retract	yes	*	no
supports clause/decompile/listing	yes	yes	no
supports debuggers	yes	yes	no
builtnoin to run time kernel	yes	yes	no
provides source code security	no	no	yes

* because it is possible to remove and reapply hashing at any time, a hashed relation can be modified simply by converting it back into an incremental relation, performing

the desired updates, and then rehashing it.

The Optimising Compiler

The optimising compiler provided with *WIN-PROLOG* can be used to increase the speed of incrementally optimized Prolog programs. It does this by creating switch statements and index tables that index not only on the first argument but on multiple arguments. Prolog code that has been optimised using the optimising compiler cannot be listed, edited or otherwise viewed. For more details on these predicates please refer to the 'Technical Reference'.

Programs can be optimised on a predicate basis, using the predicate *optimize/1*, or on a file to file basis, using the predicate *optimize_files/1*.

Predicates Related to The Optimising Compiler

index/2	*declare multiple argument indexes*
optimize/1	*optimize a static predicate*
optimize_files/1	*file to file optimization of code*

First Argument Indexing

When a static Prolog procedure is called and the first argument of the call is instantiated, the type of that argument is used to select the first clause to solve the procedure call. A clause whose first argument is not of the same type as the first argument of the procedure call will not be considered (unless it is a variable). In the case of a procedure call whose first argument is an integer or atom, the value of the argument is also used to select clauses. A clause whose first argument is a variable will always be considered to solve a procedure call (regardless of the first argument in the call).

For example, given the program:

```
interpret(1) :- process_1.
interpret(2) :- process_2.
interpret(3) :- process_3.
```

First argument indexing will give fast access to the different cases of the "interpret" program. For example, when trying to solve the call:

```
interpret(2).
```

only the second clause will be selected.

First argument indexing is also used when trying to resolve a procedure call on backtracking.

The use of first argument indexing improves program efficiency in a number of ways:

Firstly, it reduces unnecessary growth of the backtrack stack. The backtrack stack will only grow when there are alternative clauses whose first argument may match with the first argument in a procedure call. For example consider the program:

```
test(a).
test(b).
test(c).
```

The call:

```
test(b).
```

will not generate any backtrack points as there is only one clause that can possibly match the call.

Without first argument indexing, a Prolog engine would try to use the first clause to solve the call. A backtrack frame would be generated which points at the second clause for "test". The selected clause would fail (because 'a' does not unify with 'b'), causing the system to backtrack to the second clause for "test". Before calling this clause another backtrack point would be created, this one pointing at the third clause for "test" (even though this clause will not match the procedure call).

The second advantage of ***WIN-PROLOG***'s indexing mechanism is that it may make a procedure deterministic when called with an instantiated first argument. As we have just seen, this means that the backtrack stack will not grow for deterministic uses of the procedure. It also means that when the procedure exits, it may be possible to pop the corresponding call frame from the call stack (provided any sub-goals were also deterministic).

The third advantage of first argument indexing is that it may make a program tail recursive. For example, the following definition of append is tail recursive, provided the first argument is given when it is called:

```
append([], List, List).
append([Head|Tail], List1, [Head|List2]) :-
        append(Tail, List1, List2).
```

First argument indexing means that when the recursive call in the first clause is executed, the second clause will not be tried on backtracking.

The first argument indexing in the incremental compiler will partition variables, atoms, integers, real numbers, empty lists, lists and compound terms. This means that the atom 'fred' will be differentiated from the atom 'john', the integer 15 will be differentiated from the integer 59 and so on…

A compound term, 'f(x)', is also distinguished from other compound terms with different names ('g(a)', 'h(a,b,c)' etc…).

Note: The string datatype is not partitioned, so the argument:

```
`fred`
```

will not be differentiated from the string:

```
`john` .
```

The float datatype is also not partitioned, so the argument:

```
1.234
```

will not be differentiated from the float:

```
6.789 .
```

Multiple Argument Indexing in the Optimising Compiler

When using the optimising compiler you can specify the number of arguments that you wish a particular predicate to be indexed on using the predicate *index/2*. For example, given a customer database with three arguments 'Surname', 'Age' and 'Address':

```
customer(smith,35,'97 Bloggs Lane').
customer(smith,24,'52 Raindrop Crescent').
customer(smith,11,'2 Laneger Street').
customer(smith,89,'15 Twining Road').
customer(jones,102,'8a Guildhall Crescent').
customer(jones,43,'23 Light Avenue').
customer(jones,27,'894 Merton Road').
customer(jones,19,'The Mews Dripping Lane').
```

You could query this database in the following way:

```
?- customer(jones, Age, '894 Merton Road').
```

Because the incremental compiler has first argument indexing, ***WIN-PROLOG***'s inference engine will go *directly* to the first 'customer' clause whose first argument is 'jones' and then step through each consecutive clause until the third argument is matched. At this point the Age variable will be instantiated to 27.

You could speed up this process by setting the optimising compiler to index this database on both the surname and the address arguments using the following call.

```
?- index(customer/3, [1,3]).
```

Then if you optimise the database:

```
?- optimize(customer/3).
```

The query:

```
?- customer(jones, Age, '894 Merton Road').
```

will be matched *directly* against the seventh clause in the 'customer' database. In this example the query, after the database has been optimised, will take approximately half the time to find a solution as the same query asked before the database was optimised and the benefits will be even greater on larger and more complex examples.

Checking the Index of a Predicate

The type of index set on a predicate is a property of that predicate and is found using *predicate_property/2*. For example the following call will test the index set on the above `customer`/3 predicate.

- ?- predicate_property(customer(_,_,_), index(Index)).
  Index = [1,3]

Data Compression and Encryption

Two useful features of *WIN-PROLOG* are its LZSS data compression and decompression routines and its MZSS data encryption and decryption routines. These are available for general use in the *stuff/3, fluff/3, encode/2* and *decode/2* predicates.

Predicates related to data compression and encryption

stuff/3	*compress the data in the current input stream to the current output stream*
fluff/3	*decompress the data in the current input stream to the current output stream*
encode/2	*encode a data stream using MZSS encryption*
decode/2	*decode a data stream from input to output using MZSS decryption*

Abort LZSS Compression

As its name suggests, LZSS compression is based on Lempel-Ziv compression, and is a modified version of the original LZ77 algorithm. In this algorithm, a sliding window is maintained over recently output data, while a look-ahead buffer peeks into the stream of data yet to be compressed. The contents of the look-ahead are compared with all locations in the sliding window, and if a match of two or more characters is found, the address and length of the match within the window, rather than the characters themselves, is output. Depending upon the sizes of sliding window and look-ahead buffer, compression ratios of up to 64:1 are theoretically possible for highly patterned data; in practice, ratios of 2:1 to 4:1 are more usual.

The stuff/3 and fluff/3 Predicates

The predicates which implement LZSS compression and decompression are called *stuff/3* and *fluff/3* respectively. Both of these work by taking their input from the current input stream, and emitting their output to the current output stream. Their three arguments specify the size of sliding window (look-ahead is computed from this), the total number of raw (uncompressed) bytes processed, and the total number of compressed bytes processed. By experimenting with the sliding window size and comparing the last two values, it should be easy to determine the optimimum setting for whichever type of data you want to compress.

About MZSS Encryption

As its name suggests, MZSS encryption makes use of a Marsaglia/Zaman pseudo random number generator, which has the primary benefit of offering a very large key size (1185 bits, compared with just 64 bits in ***WIN-PROLOG***'s existing linear-congruential PRANG!). The MZ/PRANG is seeded by a user-specified password of up to 148 characters, and successive numbers are then combined (xor) with the plaintext in order to encrypt it, or with the cyphertext in order to decrypt it. Several special features of MZSS encryption make it especially secure.

The encode/2 and decode/2 Predicates

The predicates which implement MZSS encryption and decryption are called *encode/2* and *decode/2* respectively. Both of these work by taking their input from the current input stream, and emitting their output to the current output stream. Their two arguments specify the password key and the total number of raw (unencrypted) bytes processed.

Built-in Dialogs

WIN-PROLOG has a built-in dialog for displaying messages and getting a standard response from the user. For more details on these predicates please refer to the 'Technical Reference'.

Programs can be optimised on a predicate basis, using the predicate *optimize/1*, or on a file to file basis, using the predicate *optimize_files/1*.

Predicates Related to Dialogs

abtbox/3	*display the standard about box dialog*
'?CHANGE?'/3	*user-defined hook for handling change box messages*
change_hook/3	*system handler for the change dialog*
chgbox/3	*display the system change box dialog*
'?FIND?'/3	*user-defined hook for handling find box messages*
find_hook/3	*system hook for handling find box messages*
fndbox/2	*display the system find box dialog*
fntbox/3	*invokes the font selection dialog*
message_box/3	*create a message box and return a response*
msgbox/4	*create a message box and return a response*
opnbox/5	*display the "open file" common dialog box*
prnbox/4	*display the "print/print setup" common dialog box*
savbox/5	*display the "save as" common dialog box*

sttbox/2	*create or close a status box and return immediately*

Message Box

The predicate *message_box/3* can be used to create a dialog with some given text, a given set of standard buttons and will return a standard response from the user. For example, the following query will create a dialog with two buttons ('yes' and 'no') and the text `Would you like some tea?`. When the user has responded to the question it will return in its third argument the answer which will be one of the atoms 'yes' or 'no'.

```
?- message_box(yesno,`Would you like some tea?`,Ans).
```

This predicate is compatible between all three platforms.

Programmable Hooks and Handlers

During the running of *WIN-PROLOG* there are a number of set entry points where special user-defined programs can modify the normal behaviour of the system. These programs are referred to as "programmable hooks", where each hook has a default functor name that reflects the behaviour to be modified. Most of these hooks allow access at points that are crucial to the normal running of the *WIN-PROLOG* environment, so for each hook there is a corresponding built-in hook predicate that can be used to hand control back to *WIN-PROLOG* and allow its normal behaviour to continue.

WIN-PROLOG contains hooks that allow access to the behaviour of the system at the following places:

- When an error occurs.
- When a keyboard break occurs.
- When the debugger is called.
- When an abort happens.

All of the programmable hooks have a functor name and a set arity that is known to *WIN-PROLOG*. At each point where a programmable hook can take control, a check is made to see if a user-defined program of this name exists. The hook names and their corresponding built-in equivalents are shown in the table below. The descriptions in this chapter use the default names for the hooks.

Default Hook	Built-in Equivalent
'?ABORT?'/0	abort_hook/0
'?BREAK?'/1	break_hook/1
'?CHANGE?'/3	change_hook/3
'?DEBUG?'/1	debug_hook/1
'?ERROR?'/2	error_hook/2
'?FIND?'/3	find_hook/3
'?MESSAGE?'/4	message_hook/4
'?TIMER?'/3	timer_hook/3

Table 16 - Programmable hook names and their built-in equivalents

---

WIN-PROLOG Hooks

The following hooks allow access to error reporting, program breaks, debugging programs and aborting programs.

The programmable hooks all come into effect when a Prolog goal is interrupted. The error, break and debug hooks include the actual goal that was being interrupted as one of their arguments. If when one of these hooks is invoked you bind any of the variables in the goal and the hook succeeds, this binding will be passed back to the interrupted program. For example, a definition could be made for the '?ERROR?'/2 hook, which catches the case where the low-level read predicate *eread/1* generates an end of file error. The hook could then bind the argument of *eread/1* to the atom 'the end'.

```
'?ERROR?'(48,eread(X)):-
        X = 'the end',
        !.

'?ERROR?'(X,Y):-
        error_hook(X,Y).
```

When a hook catches an interrupt in a program in this way, the program behaves as if the hook actually appears in-line in the interrupted program. For example a program could have the following code:

```
foo :-
        bar,
        eread(X),
        sux(X).
```

Then, using the definition of the *'?ERROR?'/2* hook shown above, if the call to *eread/1* generates error *48 - End Of File*, the code that is run will be equivalent to:

```
foo :-
        bar,
        '?ERROR?'(_,eread(X)),
        sux(X).
```

where X will be bound to the atom 'the end'. Note: if the hook fails it is equivalent to the in-line call failing which causes backtracking.

Error Hook

The error hook is called whenever an error is thrown to the system (for more information on the reporting of errors see *catch/2* and *throw/2*). The only errors that cannot be caught by this hook are the memory errors (such as the program space full error). This is due to the case where running any Prolog goal could cause a memory error, which would include the Prolog goal that defined the error hook, thus causing a perpetual unbreakable error loop.

The error hook should be a Prolog program of the form:

```
'?ERROR?'( Number, Goal) :-
        Body...
```

where *Number* is the number of the error that was thrown and *Goal* is the goal that threw the error.

To pass errors through to ***LPA-PROLOG***'s default error handler, you can call *error_hook/2*, which processes the error as normal. For example, the following definition for the error hook will filter out 'predicate not defined errors', causing the undefined predicate to fail. Any other error will be passed to the default error hook.

```
'?ERROR?'( 20,Goal):-
        !,
        fail.
'?ERROR?'(ErrNum,Goal):-
        error_hook(ErrNum,Goal).
```

Keyboard Break Hook

When a ***WIN-PROLOG*** program is running you can request that the program is interrupted by pressing some special key combination on the keyboard. In ***WIN-PROLOG*** and ***DOS-PROLOG*** this happens when the <ctrl-break> key is hit.

The break hook should be a Prolog program of the form:

```
'?BREAK?'( Goal) :-
        Body...
```

where *Goal* is the goal that was interrupted.

To allow ***WIN-PROLOG*** to process break messages in its default manner you should call *break_hook/1*. This puts up a dialog which displays the interrupted goal, and offers the options of either resuming the evaluation or aborting.

In a stand-alone application you might not want end users to be able to break into it while it is running. The following example disables breaks by simply continuing with any call that is interrupted.

```
'?BREAK?'(Call) :-
        Call.
```

Debug Hook

The debug hook is invoked whenever a spied predicate is called and debugging mode is set to "debug".

The debug hook should be a Prolog program of the form:

```
'?DEBUG?'( Goal) :-
        Body...
```

where *Goal* is the goal that was called and had a spypoint set.

The default debugger is *debug_hook/1*. This passes control to the currently set system debugger.

The following program for the debug hook will count the number of times a spied predicate is called without calling the system debugger.

```
'?DEBUG?'(Call) :-
        countcall(Call),
        force(Call).

countcall(Goal):-
        def(callnumber,2,_),
        retract(callnumber(Goal,Number)),
        NewNumber is Number + 1,
        assert(callnumber(Goal,NewNumber)),
        !.

countcall(Goal):-
        assert(callnumber(Goal,1)).
```

When your program has finished, to find the number of times each spied predicate was called, you could make the following call:

```
?- forall(      callnumber(Goal,Number),
                (
                  write(Goal-Number),
                  nl,
                  retract(callnumber(Goal,Number))
                )
        ).
```

Abort Hook

The abort hook is called every time a call to the predicate *abort/0* is made. Note that *abort/0* is called by the default error hook. The hook should be a Prolog program of the form:

```
'?ABORT?' :-
        Body ....
```

To allow ***WIN-PROLOG*** to process abort conditions in its default manner you should call *abort_hook/0*.

The following abort handler, upon an abort, removes any currently defined facts associated with the system and then restarts the system from the beginning:

```
'?ABORT?' :-
        write('restarting application...'),
        abolish(runtime_facts/1),
        start_mysystem.
```

where the predicate "runtime_facts/1" are facts that have been asserted during the running of the application, and the user-defined program "start_mysystem/0" restarts the user's application from the beginning.

Appendix A - System Operators

Table 17 shows the complete set of built-in operators available in ***WIN-PROLOG***.

Operator	Precedence	Type
volatile	1150	fx
?-	1200	fx
one	900	fx
+	500	fx
-	500	fx
initialization	1150	fx
dynamic	1150	fx
public	1150	fx
mode	1150	fx
:-	1200	fx
multifile	1150	fx
meta_predicate	1150	fx
not	900	fy
spy	900	fy
nospy	900	fy
\+	900	fy
@>=	700	xfx
<	700	xfx
@<	700	xfx
=	700	xfx
@=	700	xfx

>	700	xfx
@>	700	xfx
mod	300	xfx
:-	1200	xfx
=:=	700	xfx
is	700	xfx
\=	700	xfx
=\=	700	xfx
@\=	700	xfx
=<	700	xfx
==	700	xfx
@=<	700	xfx
-->	1200	xfx
\==	700	xfx
=..	700	xfx
>=	700	xfx
,	1000	xfy
:	600	xfy
;	1100	xfy
^	200	xfy
|	1100	xfy
->	1050	xfy
~>	850	yfx
<~	850	yfx
>>	400	yfx
//	400	yfx

*	400	yfx
+	500	yfx
-	500	yfx
/	400	yfx
/\	500	yfx
\/	500	yfx
<<	400	yfx

*Table 17 - **WIN-PROLOG** built-in operators*

Index