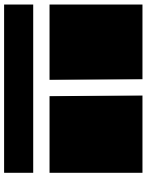
ALS Prolog
User Guide



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

The ALS Prolog Language: Standard Part

8 The Syntax of ALS Prolog

This chapter describes the syntax of ALS Prolog, which is for the most part the syntax of the ISO Prolog standard . Prolog syntax is quite simple and regular, which is a great strength.

8.1 Constants

The simplest Prolog data type is a constant, which comes in two flavors:

- atoms (sometimes called ***symbols***)
- numbers

The notion of a *constant* corresponds roughly to the notion of a *name* in a natural language. Names in natural languages refer to things (which covers a lot of ground), and constants in Prolog are be used to refer to things when the language is interpreted.

8.1.1 Numbers

Prolog uses two representations for numbers:

- integer
- floating point

When it is impossible to use an integer representation due to the size of a nominal integer , a floating point representation can be used instead. This means that extremely large integers may actually require the extended precision of a floating point value. Any operation involving integers, such as a call to `is/2`, will first attempt to usean integer representation for the result, and will use a floating point value only when necessary. This type *coercion* is carried out consistently within the Prolog system.

There is no automatic conversion of floating point numbers into integers[1].

1. In earlier versions of ALS Prolog, if a floating point number had no fractional part, and was within the range of an ALS Prolog integer, it would be represented internally as an integer. However, the ISO Prolog standard now forbids this.

Integers

The textual representation of an integer consists of a sequence of one or more digits (`0` through `9`) optionally preceeded by a '`-`' to signify a negative number. The parser assumes that all integers are written using base ten, unless the special binary, octal, or hexadecimal notation is used.

The hexadecimal notation is a `0x` followed by a sequence of valid hexadecimal digits. The following are valid hexadecimal digits:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

The octal notation is a 0o followed by a sequence of valid octal digits. The octal digits are:

```
0 1 2 3 4 5 6 7
```

The binary notation is a 0b follwed by a sequence of 0's and 1's.

Here are some examples of integers:

```
0   4532   -273  0000001  0x1fff  0b1001 0o123
```

It is important to note that a term of the form `+5` is not an integer, but rather a structured term.

Floating point numbers

Floating point numbers are slightly more complex than integers in that they may have either a fractional part, an exponent, or both. A ***fractional floating point number*** consists of a sequence of one or more numeric characters, followed by a dot ('`.`'), in turn followed by another sequence of one or more numeric characters; the entire expression may optionally be preceded by a '`-`'. Here are some examples of floating point numbers:

```
0.0   3.1415927  -3.4  000023.540000
```

You can also specify an exponent using ***scientific notation***. An exponent is either an `e` or an `E` followed by an optional '`-`', signifying a negative exponent, followed by a sequence of one or more numeric characters. Here are examples of floating point numbers with exponents:

```
0.1e-3   10E99  -44.66e-88  0E-0
```

ASCII Codes

ASCII (American Standard Code for Information Interchange) codes are small integers between `0` and `255` inclusive that represent characters. The parser will translate any printable character into its corresponding ASCII integer. In order to get the ASCII code for a character, preceed the character by the characters `0'`. For example, the code for the characters 'A', '8', and '%' would be given by:

```
0'A   0'8   0'%
```

In addition, the ANSI C-style octal and hex forms expression can be used. Thus, all of the expressions below denote the number 65:

```
0'A   0'\101   0'\x41
```

Table 1 (*Example ASCI Code Sequences.*) below contains some example ASCII code specifications.

Expression	Octal Expression	Hex Expression	ASCI Code (Decimal)	Character
0'A	0'\101	0'\x41	65	Upper case A
0'c	0'\143	0'\x63	99	Lower case c
0'~	0'\176	0'\x7e	126	Tilde character

Table 1. Example ASCI Code Sequences.

There also exists a small collection of symbolic control characters which can be thought of as synonyms for certain of the ASCI control character codes. These are presented in Table 3 (*Symbolic Control Characters.*)

Table 2:

Expression	Octal Expression	Hex Expression	ASCI Code (Decimal)	Character
0'\a	0'\007	0'\x7	7	alert ('bell')
0'\b	0'\010	0'x\8	8	backspace
0'\f	0'\014	0'\xC	12	form feed

Table 2:

Expression	Octal Expression	Hex Expression	ASCI Code (Decimal)	Character
0'\n	0'\012	0'\xA	10	new line
0'\r	0'\015	0'\xD	13	return
0'\t	0'\011	0'\x9	9	horizontal tab
0'\v	0'\147	0'\x77	119	vertical tab

Table 3. Symbolic Control Characters.

Chapter 31 (*ASCII Table*) in the Reference Manuals presents the full ASCII character set.

8.1.2 Atoms

An atom is a sequence of characters that are parsed together as a constant.

Alphanumeric atoms

An alphanumeric atom is a sequence of characters that begins with a lower case letter, and is followed by zero or more alphanumeric characters, possibly including '_'.[1] Here are some examples of alphanumeric atoms:

```
foobar123   zIPPY   bread_and_butter   money
```

Quoted atoms

A quoted atom is formed by placing any sequence of characters between single quotes (''). A single quote can be included in the text of the atom by using two consecutive single quotes for each one desired, or by prefixing the embedded single quote with the backslash (\) escape character. The following are all quoted atoms:

```
'any char will do'   '$*#!#@%#*'   'Can''t miss'
```

1. Earlier versions of ALS Prolog allowed presence of the '$' symbol in unquoted atoms. This has been dropped as part of conformance to the Prolog Standard.

```
'Can\'t miss'    '99999'
```

If the characters that compose a quoted atom can be interpreted as an atom when they occur without the enclosing single quotes, then it is not necessary to use the quoted form. However, if the atom contains characters that aren't allowed in a simple atom, then the quotes are required. Note that the last example above is an atom whose print name is `99999`, not the integer `99999`.

Quoted atoms can span multiple lines, but in this case the end of each such line must be preceeded by the backslash escape character, as in the following example of an atom:

```
'We are the stars which sing. \
We sing with our light; \
We are the birds of fire, \
We fly over the sky. \
    -- Algonquin poem.'
```

Special atoms

A special atom is any sequence of characters from the following set:

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

In addition, the atoms, `[]`, `!`, `;` and `,` are considered to be special atoms. Some other examples of special atoms are:

```
+=    &&    @>=    ==    <---------
```

Most special atoms are automatically read as quoted atoms unless they have been declared as operators (See "Operators" on page 57).

8.2 Variables

A variable consists of either a `_` (underbar character) or an upper case letter, followed by a sequence of alphanumeric characters and dollar signs. Here are some variables:

```
Variable X123a    _a$bc    _123    _
```

8.3 Compund Terms

A compound term is consists of a symbolic constant, called a functor, followed by a left parenthesis followed by one or more terms separated by commas, followed by a right parenthesis. The number of terms separated by commas enclosed in the parentheses is called the *arity* of the structure. For example, the compound term

```
f(a,b(X),y)
```

has arity 3.

8.4 Curly Braces

Instead of prefixing a structured term with a functor, the curly brace notation allows a sequence of terms, separated by commas, to be grouped together in a comma list with '`{}`' as the principal functor. For example,

```
{all,the,young,dudes}
```

parses internally into:

```
'{}'((all,the,young,dudes))
```

8.5 Lists

The simplest list is the empty list, represented by the atom '`[]`'. Any other list is a structured term with `./2` as principal functor and whose second argument is a list. Lists can be written by using '`.`' explicitly as a functor, or using the special *list* notation.

A list using list notation is written as a `[` followed by the successive first arguments of all the sublists in order seperated by commas, followed by `]`. The following are all different ways of writing the same list:

```
a.b.c.[]        [a,b,c]        '.'(a,'.'(b,'.'(c,[])))
```

Unless specified, the last tail of a list is assumed to be `[]`. A tail of a list can be specified explicitly by using `|`, as in these examples:

```
[a|X]    [1,2,3|[]]      [Head|Tail]
```

The list notation for lists is preferrable to using '`.`' explicitly because the dot is also

used in floating point numbers and to signal termination of input terms.

8.6 Strings

A string is any sequence of characters enclosed in double quotes ("). The parser automatically translates any string into the list of ASCII codes that corresponds to the characters between the quotes. For example, the string

```
"It's a dog's life"
```

is translated into

```
[73,116,39,115,32,97,32,100,111,103,39,115,32,108,10
  5,102,101]
```

Double quotes can be embedded in strings by either repeating the double quote or by using the backslash escape character before the embedded ", as for example in

```
"She said, ""hi.""".
"She said, \"hi.\"".
```

8.7 Operators

The prefix functor notation is convenient for writing terms with many arguments. However, Prolog allows a program to define a more readable syntax for structured terms with one or arguments. For example, the parser recognizes the text

```
a+b+c
```

as an expression representing

```
+(+(a,b),c)
```

because the special atom + is declared as an infix operator. Infix operators are written between their two arguments. For the other operator types, prefix and postfix, the operator (functor) is written before (prefix) or after (postfix) the single argument to the term.

What Makes an Operator?

Operators are either alphanumeric atoms or special atoms which have a corresponding *precedence* and *associativity*. The associativity is sometimes referred to as the

type of an operator. Operators may be declared by using the `op/3` builtin.

Precedences range from 1 to 1200 with the lower precedences having the tightest binding. Another way of looking at this is that in an expression such as `1*X+Y`, the operator with the highest precedence will be the principal functor. So `1*X+Y` is equivalent to `'+'('*'(1,X),Y)` because the '`*`' binds tighter than the '+'.

The types of operators are named

`fx`, `fy`, `xf`, `yf`, `xfx`, `yfx`, and `xfy` ,

where the'`f`' shows the position of the operator. Hence, `fx` and `fy` indicate prefix operators, `yf`, and `xf` indicate postfix operators, and `xfx`, `yfx`, and `xfy` indicate infix operators. An '`x`' indicates that the operator will not associate with operators of the same or greater precedence, while a '`y`' indicates that it will associate with operators of the same or lower precedence, but not operators of greater precedence.

The default or predefined operators are listed in the following tables:

Table 4: Predefined Binary Operators in ALS Prolog

Operator	Specifier	Precedence	Operator	Specifier	Precedence
:-	xfx	1200	=:=	xfx	700
-->	xfx	1200	=\=	xfx	700
==>	xfy	1200	<	xfx	700
when	xfx	1190	=<	xfx	700
where	xfx	1180	>	xfx	700
with	xfx	1170	>=	xfx	700
if	xfx	1160	:=	xfy	600
;	xfy	1100	+	yfx	500
|	xfy	1100	-	yfx	500

Table 4: Predefined Binary Operators in ALS Prolog

Operator	Specifier	Precedence	Operator	Specifier	Precedence
->	xfy	1050	/\	yfx	500
,	xfy	1000	\/	yfx	500
:	xfy	950	xor	yfx	500
.	xfy	800	or	yfx	500
=	xfx	700	and	yfx	500
\=	xfx	700	*	yfx	400
==	xfx	700	/	yfx	400
\==	xfx	700	//	yfx	400
@<	xfx	700	div	yfx	400
@=<	xfx	700	rem	yfx	400
@>	xfx	700	mod	yfx	400
@>=	xfx	700	<<	yfx	400
=..	xfx	700	>>	yfx	400
is	xfx	700	**	xfx	200
			^	xfy	200

Table 5: Predefined Prefix Operators in ALS Prolog

Operator	Specifier	Precedence	Operator	Specifier	Precedence
:-	fx	1200	nospy	fx	800
?-	fx	1200	-	fy	200
vi	fx	1125	+	fy	200
edit	fx	1125	\	fy	200
ls	fx	1125	export	fx	1200
cd	fx	1125	use	fx	1200
dir	fx	1125	module	fx	1200
not	fx	900	``	fx	925
\+	fx	900	`	fx	930
trace	fx	800	~	fy	300
spy	fx	800			

Special Cases

It is possible to declare an operator via `op/3` that can never be parsed. Even though quoted atoms can be assigned a precedence and associativity, the parser will only interpret alphanumeric atoms or special atoms as operators.

White space

White space, or *layout characters*, refers to the part of source code, data, and goals that is not made up of readable characters. The term white space comes from the fact that these unreadable characters appear white when source code is printed on a sheet of white paper. White space is any sequence of spaces, tabs, or new lines. Generally speaking, white space has little meaning to the parser. It is occasionally important for recognizing full stops, and for delimiting constructs which, if they

were run together, would not be recognizable as separate constructs. There are also places where additional white space is either inappropriate or changes the meaning of the text. For example, you can't embed a space in a number.

8.8 Comments

Comments can be put anywhere white space can occur. Comments can take one of two forms:

1. A line comment: anything following a percent sign (%) is ignored until the end of line.
2. A block comment: anything enclosed in a '`/* */`' pair is ignored. Block comments may span many lines if desired. Block comments may be nested, thus allowing commented code to be commented out.

```
/*
 *
 *  /*
 *   *    This is one way to use block comments
 *   */
 *
 */

connected(footbone, legbone).

/*    Here's another    */

connected(headbone, neckbone). % line comments can
connected(dogbone, fishbone).  % look good
                               % next to code that
                               % you write
```

8.9 Preprocessor Directives

ALS Prolog supports ***preprocessor directives*** which can affect the text at the time the program is compiler (or loaded into an image). These expressions include the

following[1]:

```
#include   #if   #else  #elif   #endif
```

Each of these must occur at the beginning of a line of program text. Each of `#include`, `#if`, and `#elif` must be followed by a Prolog term, but each of `#else` and `#endif` must stand on a line by themselves. The `#include` directive should be followed by a Prolog double quoted string, intended to name a file:

```
#include "/mydir/foo.pro"
```

No fullstop (.) should follow this expression, nor the expressions following `#if` and `#elif.` The expression following `#if` or `#elif` can be an arbitrary Prolog term.

The expressions `#if`, `#else, #elif, #endif` must be organized as conditionals in a manner similar to their use in C programs. Thus, the first expression occurring must be an `#if`, and the last must be an `#endif`. Between them there can be zero or more occurrences of `#else` and `#elif`. There can be at most one occurrence of `#else` between a given `#if` ... `#endif` pair, and it must follow all of the zero or more occurrences of `#elif` between the same pair.

Preprocessor directive semantics appears in Section 9.3 (*Preprocessor Directives.*) .

1. This list might be extended in the future. The most notable candidate would be `#define.`

9 Prolog Source Code

Just like most other computer languages, Prolog allows you to store programs and data in text files. The builtin predicates `consult/1`, `reconsult/1`, `read/1`, and their variants will translate the textual representation for programs and data into the internal representations used by the Prolog system. This section describes what kinds of syntactic objects can appear in source files and how they are interpreted.

9.1 Source Terms

Every Prolog source file must be a sequence of zero or more Prolog terms, each term followed by a period (`.`) and a white space character. A period followed by a white space character is called a *full stop*. Full stops are needed in source files to show where one term ends and another begins. Each term in a file is treated as a closed logical formula. This means that even though two seperate terms have variable names in common, each term's variables are actually distinct from the variables in any other term. Most variables are quantified once for each term. However, there is a special variable, the anonymous variable (written '_ '), which is quantified for each occurrance. This means that within a single term, every occurrance of '_' is a different variable.

9.1.1 Rules

A *rule* is the most common programming construct in Prolog. A rule says that a particular property holds if a conjunction of properties holds. Rules are always written with `:-/2` as the ***principal functo***r. The first argument of the `:-` is called the *head*, and the second argument is called the *body*. Here are some examples of rules:

```
a(X) :- b(X).
blt :- bacon, lettuce, tomato.
test(A,B,C) :- cond1(A,B), cond2(B,C).
```

`consult/1` and `reconsult/1` load rules (and facts - see below) from a source file into the internal run-time Prolog database in the order they occur in the source file.

9.1.2 Facts

A ***fact*** is any term which is not a rule and which cannot be interpreted as a directive or declaration. For example,

```
module mymodule.
```

would not be interpreted as a fact since it is a module declaration - see Chapter 10 (*Modules*) . More specifically, a fact is any term that cannot be interpreted as a declaration and whose principal functor is not `:-/1`, `?-/1`, or `:-/2`. One way to understand a fact is to say that it is a rule without a body, or a rule with a trivial body (one that is always true). These are example facts:

```
mortal(socrates).
big(ben).
identical(X,X).
```

As with rules, `consult`/1 and `reconsult`/1 load facts (and rules)from a source file into the internal database in the order they occur in the source file.

9.1.3 Commands and Queries

A *command* or *directive* is any term whose principal functor is `:-/1`. Queries are terms whose principal functor is `?-/1`. The single argument to a command or query is a conjunction of goals to be run when `consult` or `reconsult` encounters the construct in a source file.

Commands are often used to add operator declarations to the parser, or to implement command files. Queries are just like commands except they print out `yes` or `no` depending on whether the query succeeded for failed.

Commands are silent unless the command fails, or unless some goal inside the command writes to the current output output stream. If a command fails during the process of consulting a file, a warning message is written to standard output, which is usually the screen or console window.

Here are some examples of commands:

```
:- op(300, xfx, #).
:- [file1], [file2],
       write('Files 1&2 have been loaded').
```

```
:- initializeProgram, topLevelGoal.
```

9.1.4 Declarations

Declarations are terms that have a special interpretation when seen by `consult` or `reconsult`. Here are some example declarations:

```
use builtins.
export a/1, b/2, c/3.
module foobar.
```

9.2 Program Files

Program files are sequences of source terms that are meant to be read in by `consult/1` or `reconsult/1`, which interpret the terms as either clauses, declarations, commands, or queries.

9.2.1 Consulting Program Files

To consult a file means to read the file, load the file's clauses into the internal Prolog database, and execute the commands or directives occurring in the file. Reconsulting a file causes part or all of the current definitions in the internal database for procedures which occur in the file to be discarded and the new ones (from the file) to be loaded, as well as executing commands or directives in the file. See Chapter 16 (*Prolog Builtins: Non-I/O*) `consult/1`.

A file is consulted by the goal

```
?- consult(filename).
```

or reconsulted by the goal

```
?- reconsult(filename).
```

Several files can be consulted or reconsulted at once by enclosing the file names in list brackets, as in

```
?- [file1,file2,file3].
```

By default, files listed this way (inside list brackets) are *reconsulted*.[1] To insist that one or more files in such a list be consulted (which might cause some of the clauses

from the files to be doubled in memory), prefix a '+' to the filename, as in:

```
?- [file1,+file2,file3].
```

In this case, `file2` will be consulted instead of reconsulted. For consistency and backwards compatibility, one can also prefix a '-' to indicate that the file should be reconsulted, even though this is redundant:

```
?- [file1,+file2,-file3].
```

When any of these consult goals are presented, first the terms from `file1` are processed, then the terms from `file2`, and finally the terms from `file3`. It is permitted that clauses for the same procedure to occur in more than one file being consulted. In this case, clauses from the earlier file are listed in the internal database before clauses from the later file. Thus, if both `file1` and `file2` contain clauses for procedure `p`, those from `file1` will be listed in the internal database before those from `file2`. Clauses in the internal database are 'tagged' with the file from which they originated. When a file is reconsulted, only those clauses in memory which are tagged as originating from that file will be discarded at the start of the reconsult operation. Thus, suppose that both `file1` and `file3` contain clauses for the procedure `p`, and that we initally perform

```
?- [file1,file2,file3].
```

Then suppose that we edit `file3`, and then perform

```
?- [file3].
```

The clauses for `p` originally loaded from `file1` will remain undisturbed. The clauses currently in memory for `p` originally from `file3` will be discarded, and the new clauses from `file3` will be loaded.

9.2.2 Using Filenames in Prolog

Note: Complete path names to files are of course quite variable across operating systems. The discussions below are only intended to describe those aspects of file names and path names which affect how ALS Prolog locates files. Examples are provided for all the operating systems supported by ALS Prolog. File names follow

1. This reverses the convention of earlier versions of ALS Prolog as well as Edinburg Prolog, but reflects the preferences of most contemporary Prolog developers.

the ordinary naming conventions of the host operating system. Thus all of the following are acceptable file names:

Unix:

```
fighter     cave.man  hack/cave.man
/usr/hack/cave.man
```

Macintosh:

```
fighter     cave.man  :hack:cave.man
usr:hack:cave.man
```

Win32:

```
fighter     cave.man  hack\cave.man
C:\usr\hack\cave.man
```

In general, file names should be enclosed in single quotes (making them quoted atoms). The exception is any file name which is acceptable as an atom by itself.

Simple file names consist of only the file name, or a file name together with an extension. All others are *complex file names*. Absolute path names provide a complete description of the location of a file in the file system, while relative path names provide a description of a file's location relative to the current directory. Simple file names are interpreted as relative path names.

The way that the program-loading predicates react to the different kinds of path names is described below. In general, however, the loading predicates attempt to determine whether a file exists, and if so, they load the clauses from the file. If the file does not exist, the loading predicates raise an error exception.

If an absolute path name is used as an argument to one of the program loading predicates (`consult/1`, `reconsult/1`, etc.), that file is loaded if it exists. If the file does not exist, an error exception is raised.

If a complex relative path name or a simple file name is passed to consult, the system first attempts to locate the file relative to the current directory. In particular, for a simple file name, the system simply looks in the current directory for the file. In either case, if the file exists, it is loaded.

If the file cannot be found relative to the current directory, ALS Prolog searches for another directory containing that file. Ultimately, the directories (folders) through which ALS Prolog searches are determined by a dynamic collection of facts `searchdir/1` maintained in the `system` (or `builtins`) module. Operationally, ALS Prolog forms the list `PlacesToTry` consisting of all `D` such that

```
searchdir(D).
```

is true in the module `builtins`, putting the current directory at the head of this list, even when no `searchdir/1` assertion mentions it. Then it works its way through the elements `D` of `PlacesToTry`, attempting to locate the sought-for file relative to directory `D`. The first file located in this manner is loaded. This process is determinate: the system never restarts the search process once a file meeting the relative path desrcription has been found.

If none of the directories listed on PlacesToTry provide a path to the sought-for file, ALS Prolog locates the *alsdir* subdirectory from its own installation, and attempts to locate the file relative to two of the subdirectories, *builtins* and *shared*, which are found in *alsdir*.

If none of these directories provides a means of locating a file with the the original complex relative path name or simple file name, the system raises an error exception.

The facts `searchdir/1` in module `builtins` can be manipulated by a user program or by the user at the console. However, ALS Prolog provides several automatic facilities for installing these facts.

- On Unix and Windows, if the ALSPATH environment variable is set , the entries from this are used to create `searchdir/1` assertions.
- If ALS Prolog was started from the command line, any ‘-s’ switches on the command line will cause `searchdir/1`assertions to be added.

Thus, the directories which will be search appear as follows:

1. First, the current directory is searched.
2. Next, any directories appearing as ‘-s’ command line switches are searched, in the order they appear from left to right on the command line.

3. Next, any directories appearing in an ALSPATH environment variable are searched, in the order they appear in the variable statement.

4. The subdirectory *builtins* of *alsdir* is searched.

5. The subdirectory *shared* of *alsdir* is searched.

Of course, if additional `searchdir/1` have been asserted or retracted, this order will be modified. Note, in particular, that `searchdir/1` assertions for module `builtins` can be included in an ALS Prolog autoload file.

9.2.3 How are Filename Extensions treated?

For your convenience, if you have a file ending with a *.pro* or a *.pl* extension, you don't have to type the extension in calls to the program loading predicates. The following goal loads the Prolog file *wands.pro*:

```
?- consult(wands).
```

What really happens is this. On a call to load a file (simple or complex) with no extension, ALS Prolog first searches for a file with exactly that name. If found, that file (with no *.pro* extension) is loaded. If no such file is found, then ALS Prolog attempts to find a file of that name with a *.pro* extension, and following that, with a *.pl* extension. Thus the example above will load *wands.pro* only if there is no file *wands* to be found, not only in the current directory, but also in the directories on the search path described above.

Whenever ALS Prolog loads a Prolog source file, it compiles the file and immediately loads and links the resulting code in memory. If the source file had a *.pro* extension, but the call to load it omitted the *.pro* extension, ALS Prolog also creates a file on the disk containing a relocatable object version of the compiled code. On all operating systems, if the source file had a *.pro* extension, but the call to load the file omitted the *.pro* extension, a file with the same name, but the extension *.obp* is created to hold the relocatable object code. Once a relocatable object file has been created, any call to load the original file will cause the relocatable object file to be loaded instead, provided that the original source file has not been modified since the object file was created. (This is determined by the date-time stamps on the two files.) The advantage of this lies in the fact that object files load much more quickly than source files. Note that on all systems but the Macintosh, the following call will

not create an object file for *wands* :

```
?- consult('wands.pro').
```

Thus, when consulting or reconsulting a file with no extension, ALS Prolog proceed as follows:

- The system will first look for the file without any extension; if found, it will load the file as is and will not create an object file.
- If the file is not found, the system will then attach the extensions *.pro* and *.obp* and look for both of these files.
- If a *.obp* version of the file exists and is newer than the *.pro* version, then the *.obp* version is loaded.
- On the other hand if the *.pro* version is newer, then the *.pro* version is loaded and a new *.obp* version is created (which is now newer than the *.pro* version).

For the system to correctly decide which file is newer, *.pro* or *.obp* (or the resource fork on the Macintosh), the system date and time should always be set correctly. The directories in which Prolog files reside should be writeable by ALS Prolog so that *.obp* versions of Prolog source files can be produced.

ALS Prolog has facilities for controlling where these **.obp* files are placed, and correspondingly, where they are searched for when (re-)loading files.

9.2.4 Splitting up Prolog Programs

It is common practice to place lines of the form

```
:- [file1,file2].
```

in files which are being consulted. This will cause both *file1* and *file2* to also be consulted. For both the consult and reconsult operations, this directive behaves as if the text for *file1* and *file2* was placed in the file being consulted at the place where the command occurred. This facility is similar to the `#include` facility found in C. A full description of all predicates for loading programs can be found on the builtins reference page for `consult/1`.

9.3 Preprocessor Directives.

Assume that *PFile* is the name of a file being consulted into ALS Prolog. If *<Filename>* is the name of another valid Prolog source file, then the effect of the preprocessor directive

```
#include "<Filename>"
```

is to textually include the lines of *<Filename>* into *PFile* as if they had actually occurred in *PFile* at the point of the directive.

The conditional preprocessor directives `#if`, #else, `#elif`, and `#endif` behave more or less as they do for C programs. However, the expressions following the `#if` and `#elif` are taken to be Prolog goals, and are evaluated in the current environement, just as for embedded commands of the form `:- G.` Here are some examples. Let *f1.pro* be the following file:

```
:-dynamic(z/1).
%z(f).

p(a).

#if (user:z(f))
p(b).
#else
p(c).
#endif

p(ff).
```

After consulting *f1.pro*, we use listing/0 to see what happened:

```
?- listing.

% user:p/1
p(a).
p(c).
p(ff).
```

```
yes.
```

In this case, p(c) was loaded, but not p(b). Now let *f2.pro* be the following file:

```
:-dynamic(z/1).
z(f).

p(a).

#if (user:z(f))
p(b).
#else
p(c).
#endif

p(ff).
```

After consulting *f2.pro* to a clean image, we obtain the following:

```
?- listing.

% user:p/1
p(a).
p(b).
p(ff).

% user:z/1
z(f).
```

This time, p(b) was loaded instead of p(c).

10 Modules

ALS Prolog provides a module system to facilitate the creation and maintenance of large programs. The main purpose of the module system is to partition procedures into separate groups to avoid naming conflicts between those groups. The module system provides controlled access to procedures within those groups.

The ALS module system only partitions procedures, not constants. This means that the procedure `foo/2` may have different meanings in different modules, but that the constant `bar` is the same in every module.

10.1 Declaring a Module

New modules are created when the compiler sees a module declaration in a source file during a `consult` or `reconsult`. Every module has a name which must be a non-numeric constant. Here are a few valid module declarations:

```
 module dingbat.
module parser.
module compiler.
```

Following a module declaration, all clauses will be asserted into that module using `assertz` until the end of the module or until another module declaration is encountered. In addition, any commands that appear within the scope of the module will be executed from inside that module.

The end of a module is signified by an `endmod`. The following example defines the predicate `test/0` in two different modules. The definitions don't conflict with each other because they appear in different modules.

```
 module mod1.
    test :- write('Module #1'), nl.
endmod.

module mod2.
    test :- write('Module #2'), nl.
endmod.
```

Clauses which are not contained inside an explicit module declaration are added to the default module `user`.

If a module declaration is encountered for a module that already exists, the clauses appearing within that declaration are simply added to the existing contents of that module. In this way, the code for a single module can be spread across multiple files—as long as each file has the appropriate module declaration and ends with a corresponding `endmod`.

The end of a file does not signal the end of the module as shown in the following conversation with the Prolog shell:

```
 ?- [user].
Consulting user ...
module hello.
a.
b.
c.
user consulted

yes.
?- [user].
Consulting user ...
module hello.
d.
endmod.
user consulted

yes.
?- listing(hello:_).

% hello:a/0
a.
% hello:b/0
b.
% hello:c/0
c.
```

```
% hello:d/0
d.
```

If module `hello` had been closed by the EOF of the *user* file, then the `d/0` fact would have appeared in the `user` module instead of the `hello` module. Consequently, it is important to terminate a module with `endmod`. That is, `module` and `endmod` should always be used in matched pairs.

10.2 Sharing Procedures Between Modules

By default, all the procedures defined in a given module are visible only within that module. This is how *name conflicts* are avoided. However, the point of the module system is to allow controlled access to procedures defined in other modules. This task is accomplished by using *export declarations* and *use lists* . `Export` declarations render a given procedure visible outside the module in which it is defined, while `use` lists specify visibility relationships between modules. Each module has a `use` list.

10.3 Finding Procedures in Another Module

Whenever the Prolog system tries to call a procedure, it first looks for that procedure in the module where the call occured. This is done automatically, and independently of `use` list and inheritance declarations.

If the called procedure `p/n` is not defined in some module, say M, from which it is called, then the system will search the `use` list of M for a module M1 that exports the procedure (`p/n`) in question. If such a module M1 is found, then all occurrences of the procedure `p/n` in the calling module M will be 'forwarded' to the procedure `p/n` defined in the module M1. After the procedure `p/n` has been forwarded from module M to another module M1, all future calls to procedure `p/n` from within M will be automatically routed to the proper place in M1 without further intervention of the module system.

The forwarding process is determinate. That is, once a call on procedure `p/n` has been forwarded to `p/n` in module M1, even if backtracking occurs, ALS Prolog will *not* attempt to locate another module M2 containing a procedure to which p/n can be forwarded.

Finally, if no module on the `use` list for M exports the procedure in question (`p/n`), then the procedure `p/n` is undefined in M, and the call fails.

10.3.1 Export Declarations

An ***export declaration*** tells the module system that a particular predicate may be called from other modules. Here are some export declarations:

```
export translate/3.
export reduce/2, compose/3.
export a/0, b/0, c/0.
```

Export declarations can occur anywhere within a module. However, one good programming style dictates that procedures are exported just before they're defined. Another stylistic alternative is to group all the export declarations for a module together in the beginning of the module. The only restriction is that visible procedures must be exported before they can be called from another module. This can happen during the execution of a command or query inside a `consult`.

10.3.2 Use Lists

Associated with each module M is a ***use list*** of other modules where procedures not defined in the given module M may be found. `Use` lists are built by ***use declarations*** which take the forms

```
use mod
use mod1, mod2, ...
```

where `mod` is the name of the module to be used. Here are some examples:

```
use bitOps, splineOps.
use polygons.
```

Each `use` declaration adds the referenced module to the front of the existing `use` list for the module M in which the `use` declaration occurs. If there is more than one module in a given `use` declaration (as in the first example above), then the listed modules are added to the front of the existing use list in reverse order from their original order in the `use` declaration. During the forwarding process, `use` lists are always searched from left to right. This means that the most recently '`used`' mod-

ules (i.e., those whose `use` declaration was made most recently) will be searched first. Here's an example of a module with `use` declarations building a `use` list:

```
module graphics.
use bitOps, splineOps.
use polygons.

test :- drawPoly(5, 0, 0).
endmod.
```

In this example, the resulting `use` list for module `graphics` would be:

```
polygons, splineOps, bitOps
```

and this is the order in which the modules will be searched, as shown in Figure 1 (*Use List Searching*).

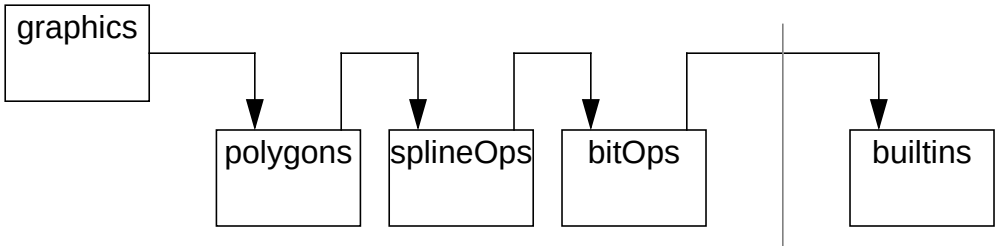


Figure 1. Use List Searching

10.4 Default Modules

Two modules, `builtins` and `user`, are automatically created when the ALS Prolog system starts up. The `builtins` module contains code that defines the standard builtin predicates of the system. All modules automatically `use` the `builtins` module, as suggested in Figure 1 (*Use List Searching*), so therefore the following declaration is implicit:

```
use builtins.
```

`user` is the default module. Any source code that is not contained within a module declaration is automatically placed in the `user` module. In addition, the `user`

module automatically uses every other module (in the order the modules are actually created), so it inherits all exported procedures.

10.5 References to Specific Modules

In addition to the `export` and `use` list conventions, the module system allows access to specific modules via the operator `:/2`, whose left hand argument is interpreted as the name of a module, and whose right hand argument is the goal to be called. In the following example, the procedure `zip` in `module1` specifically references the procedure `bar` in `module2`, even though `bar` isn't exported:

```
module module1.
    zip :- module2:bar.
endmod.
module module2.
    bar :- true.
endmod.
```

As this example demonstrates, the `export` declarations of a module aren't sacred and can be violated by `:/2`. However, good software engineering practice suggests that explict references be used only when there are compelling reasons for avoiding the `use` list and export declaration mechanism.

10.6 Nested Modules

ALS Prolog does not currently support the nesting of modules in the way Pascal does for procedures. Instead, ALS Prolog allows modules to be nested, but processes each nested module independently. Consequently, the visibility of a nested module is not limited to the module in which it is declared. The following example illustrates the effect of placing code for a module inside of another module declaration. The Prolog code below shows a declaration for a module `rhyme`, containing a three clause Prolog procedure called `animal/1`. The module is closed off by the last `endmod` declaration. In between the last two clauses of the `animal/1` procedure is the module `reason`. The module `reason` has a two clause procedure named `mineral/1`.

```
module rhyme.
```

```
export animal/1.

  animal(frog).
  animal(monkey).

       % The following module declaration
       % (temporarily) closes off the rhyme module
       % in addition to starting the reason module:
  module reason.
  export mineral/1.

       mineral(glass).
       mineral(silver).

  endmod. % reason

  animal(tiger).

endmod. % rhyme
```

Here is a conversation with the Prolog shell illustrating the effects of loading the above code:

```
?-listing.
% reason:mineral/1.
mineral(glass).
mineral(silver).
% rhyme:animal/1.
animal(frog).
animal(monkey).
animal(tiger).
```

As you can see, even though the module `reason` was nested in the module `rhyme`, the two modules are processed independently.

10.7 Facilities for Manipulating Modules

When Prolog starts up, the *current module* is `user`. This means that any queries you submit will make use of the procedures defined within `user` and the modules which are accessible from `user`'s `use` list. The current module can always be determined using `curmod/1`. It is called with an uninstantiated variable which is then bound to the current module. The predicate `modules/2` can be used to determine all of the modules currently in the system, together with their `use` lists. Assume that the following code has been consulted:

```
module m1.
use m2.
  p(a).
  p(b).
endmod.

module m2.
  q(c).
  q(d).
endmod.
```

Then the following illustrates the action of `modules/2`:

```
?- modules(X,Y).

X = user
Y = [m2,m1,builtins];

X = builtins
Y = [user];

X = m1
Y = [m2,builtins,user];

X = m2
Y = [builtins,user];
no.
```

11 Using Definite Clause Grammars

Prolog is a very powerful tool for implementing parsers and compilers. The Definite Clause Grammar (DCG) notation provides a convenient means of exploiting this power by automatically translating grammar rules into Prolog clauses. ALS Prolog translates DCG rules occurring in source files into their equivalent Prolog clauses, which are then asserted into the database. The translator itself is a Prolog program contained in the file ***dcgs.pro*** which resides in the *alsdir* directory together with the other builtins files. The sections below provide a simple sketch of the use and operation of DCGs. A more detailed presentation of the use of DCGs and the development of translators for them can be found in [bowen]. Advanced treatment of logic-based grammars is provided by [abramson], [dahl85], [dahl88], and [pereira]. DCGs have the general form

```
non-terminal --> dform1, ..., dformN.
```

where `dform1` through `dformN` are either *non-terminals*, *terminals*, or Prolog goals. Non-terminals are similar in spirit to nouns and verb phrases in natural language, while terminals resemble actual words.

```
sentence --> noun, verbPhrase.
```

The example above uses the non-terminals `noun` and `verbPhrase` to define another non-terminal called `sentence`. The intended reading of the rule is that a sentence can be formed by appending a verb phrase after a noun.

11.1 How Grammar Rules are Translated Into Clauses

The DCG expander works by adding two extra arguments (which are in fact variables) to each non-terminal. These two variables are used to pass the list of tokens to be parsed. The rule that defines a sentence would be translated into the following Prolog clause:

```
sentence(S,E) :- noun(S,I0), verbPhrase(I0,E).
```

This rule means that if `S` is a list of tokens, and if some initial sequence of `S` can be parsed as a sentence, then `E` is the list of tokens which remain after one sentence has been parsed. The first part of the sentence, the noun, is constructed from the

tokens beginning with `S` up to `I0`. The verb phrase picks up where the noun left off, and consumes tokens up to `E`. For example, if '`cat`' is a noun, and '`ran`' is a verb phrase, then the following queries will succeed:

```
?- sentence([cat,ran],[]).
?- sentence([cat,ran,away],[away]).
```

In the same manner, if `noun/2` is given the list `[cat,ran]`, it will consume `cat` and return the list `[ran]`. Similarly, `verbPhrase/2` consumes `ran` and hands back the rest of the input token list.

11.2 Writing a Grammar

Because DCG rules are translated into Prolog clauses, it is possible to have many rules that define what it means to be a sentence or a noun or a verb. If one rule can't parse the list of tokens, Prolog will fail and try the next rule. The following set of rules says that `cat`, `dog`, and `pig` are all nouns. In addition, two compound verb phrases are defined.

```
noun --> [cat].
noun --> [dog].
noun --> [pig].

verbPhrase --> verb.
verbPhrase --> verb, adverb.

verb --> [ran].
verb --> [chased].

adverb --> [away].
adverb --> [fast].
```

Here are some examples that make use of these rules:

```
?- noun([dog,chased,cat],[chased,cat]).
?- noun([pig,ate,slop],[ate,slop]).
?- sentence([cat,ran,away],[]).
?- sentence(
```

```
        [pig,ran,fast,dog,chased,cat],
        [dog,chased,cat]).
```

The following picture illustrates the consumption of the sentence:

```
[pig,ran,fast,dog].
```

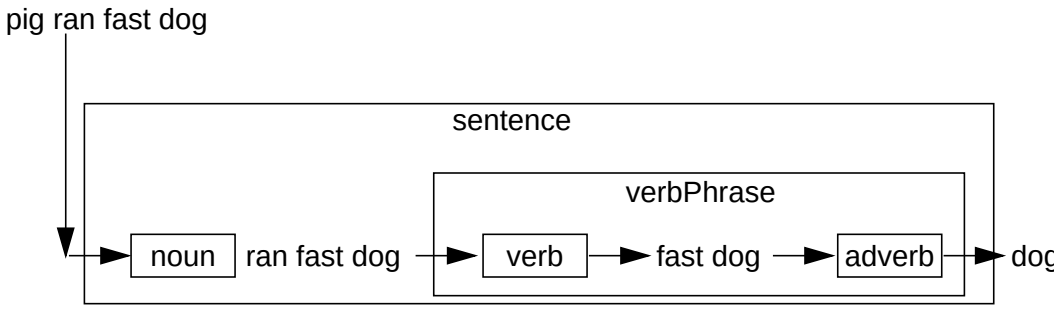


Figure 2. DCG Parsing as Filtering.

The way to read this diagram is to regard each box as a filter. The filter consumes some of the input, and allows the remaining part to pass through to the next filter. The following diagram is a tree which illustrates the structure of the parsed list:

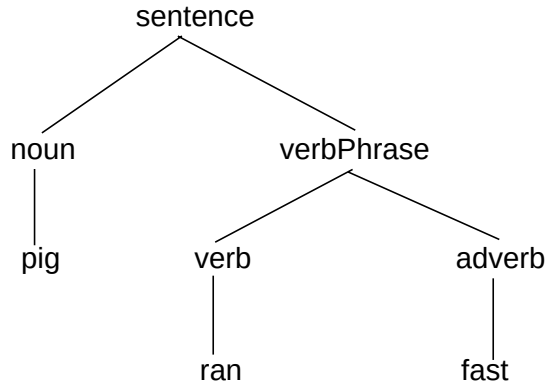


Figure 3. A Parse Tree.

DCG rules can have variables in the non-terminals which can be used to pass and return information. For instance, consider the following collection of DCG rules:

```
noun(animal(pig)) --> [pig].
noun(food(slop)) --> [slop].

nounPhrase(noun(Det,Noun)) -->
  determiner(Det),noun(Noun).

determiner(the) --> [the].
determiner(a) --> [a].
```

Here, `noun` has been defined to return a structure which would be used to differentiate between the different types of `noun`s parsed by the DCG rule. These DCGs would be translated into the following Prolog rules:

```
noun(animal(pig),[pig|E],E).
noun(food(slop),[slop|E],E).

nounPhrase(noun(Det,Noun),S,E) :-
  determiner(Det,S,I0),noun(Noun,I0,E).

determiner(the,[the|E],E).
determiner(a,[a|E],E).
```

Prolog goals can also appear within DCGs if placed between curly braces (`{` and `}`). Goals thus protected by braces are passed through untouched by the DCG expander and do not have the extra arguments added to them. For example, the following rule could be used to recognize numbers:

```
quantity(quantity(Value,Unit)) -->
  [Number],
  {convertnumber(Number,Value),number(Value),!},
  unit(Unit).
```

This rule is translated into:

```
quantity(quantity(Value,Unit),[Number|I0],E) :-
  convertnumber(Number,Value),
  number(Value),!,
  unit(Unit,I0,E).
```

Many of the builtin predicates are also left untouched whether enclosed by the curly braces or not. These are shown in Figure 4

;/2	atom/1
</2	atomic/1
=/2	fail/0
=:=/2	integer/1
=</2	is/2
=\=/2	nonvar/1
>/2	true/0
>=/2	var/1

Figure 4. Builtin Predicates Untouched by the DCG Expander.

The ALS Prolog Language: Extensions

12 Working with Uninterned Atoms

Like most symbolic programming langauges, ALS Prolog implements atoms in two different ways:

- Atoms can be *interned* which means that they have been installed in the Prolog symbol table. Atoms which have been interned are called ***symbols***.
- Atoms can also be *uninterned* which means that they have not been installed in the symbol table. These atoms are called ***UIAs*** (UnInterned Atoms).

Because UIAs are stored on the heap, they are efficiently garbage collectable. Ordinary Prolog programs cannot distinguish between interned and uninterned atoms, except for possible differences in efficiency. However, programs which must interface to other external programs can sometimes find UIAs very useful.

12.1 The Efficiency of UIAs

Symbols are entered in the symbol table only once, so comparison between atoms which are symbols is very fast. This is because only the symbol table indices need to be compared. In contrast, UIAs are stored on the heap as the sequence of characters in the atom's print name (together with header/footer information). Comparison of a symbol with a UIA, or a UIA with a UIA, is somewhat slower because the two atoms must be compared by comparing the characters in their print names.

Thus, it is desirable to store atoms as symbols if they are likely to often be compared with other atoms. This includes the functors of structures and the distinguished program constants such as ':-' or '+', etc. On the other hand, many programs contain atoms which are seldom or never compared with other atoms. Prompt messages and other output strings are good examples, as are atoms read when searching a file. These objects should usually be stored as UIAs to avoid clogging up the symbol table.

12.1.1 When is a UIA created?

ALS Prolog uses the following rules to decide whether a given occurrence of an atom should be a symbol or a UIA.

1. All functors, operators, and predicate names are put into the symbol table.
2. Atoms appearing in the text without single quotes are put in the symbol table.
3. Atoms appearing in the text enclosed in single quotes are stored as UIAs unless the string which forms the atom is already in the symbol table, or unless the first rule applies.
4. Atoms created by `name/2` are UIAs unless the string which forms the atom is already in the symbol table.

Consider the following clauses:

```
p('x',y) :- q('f','x').
p(f(y),'wombat').
p(x,'wombat').
```

Let us assume that none of `p`, `q`, `x`, `y`, `f`, or `wombat` are initially in the symbol table when these clauses are first read. Both `p` and `q` will be put into the symbol table because they are predicate names. `y` will also be put into the symbol table because it does not appear between single quotes. On the other hand, `wombat` will be stored as a UIA becasue it is surrounded by single quotes. Similarly, `x` and `f` will initially start out as UIAs because they appear in single quotes. But both of them will eventually be entered into the symbol table because `f` appears as a functor in the second clause and `x` appears unquoted in the third clause.

The rationale behind making atoms which are enclosed in single quotes into UIAs is that these sort of atoms most often appear as filenames or messages to write out. As such, they are rarely compared with other atoms.

12.2 Interning UIAs

It is sometimes desirable, under direct program control, to intern an atom which was originally stored as a UIA. This will cause all future occurrences of the atom, whether read by the parser or processed by `name/2`, to be turned into symbols. This is accomplished by using `functor/3`. Suppose that the constant `'ProgramConstant'` should be interned. This atom cannot be written in a program text without enclosing it in single quotes, because otherwise it would be read as a variable. The way to turn this into a constant is to issue the goal:

```
functor(_,'ProgramConstant',0).
```

If a large number of constants need to be interned, it may be desirable to write an intern predicate which might take the following form.

```
intern(X) :- atom(X), !, functor(_,X,0).
intern([H|T]) :- intern(H), intern(T).
```

This could then be called in the following manner:

```
intern([ 'ProgramConstant',
         'AnotherConstant',
         'YetAnotherConstant' ]).
```

All three quoted strings will be interned so that later occurrences will be stored as symbols. The `PI_forceuia()`function can also be used to intern UIAs. See `PI_forceui` in the Foreign Interface Reference.

12.3 Manipulating UIAs

Creating UIAs

There are several additional predicates which can be used to manipulate UIAs. A UIA of specific length can be created with a call to $uia_alloc/2 with the following arguments:

```
'$uia_alloc'(BufLen,UIABuf)
```

`BufLen` should be instantiated to a positive integer which represents the size (in bytes) of the UIA to allocate. The actual size of the buffer allocated will be a multiple of four greater than or equal to `BufLen`. `UIABuf` should be a variable. UIAs created with `$uia_alloc` are initially filled with zeros, and will unify with the null atom (`''`).

Modifying UIAs

Values can be inserted into a UIA buffer using a number of different routines. We will discuss two of them here: $uia_pokeb/3 and $uia_pokes/3. The modifications are destructive, and persist across backtracking.[1] `$uia_pokeb/3` is called as follows:

```
'$uia_pokeb'(UIABuf,Offset,Value)
```

`UIABuf` should be a buffer obtained from `$uia_alloc/2`. The buffer is viewed as a vector of bytes with the first byte having offset zero. `Offset` is the offset within the buffer to the place where `Value` is to be inserted. Both `Offset` and `Value` are integer,. and the byte at position `Offset` from the beginning of the buffer is changed to `Value`. Figure 5 (*Action of $uia_alloc/2.*) illustrates this action.

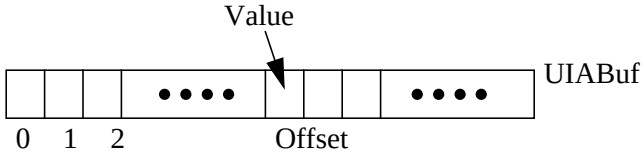


Figure 5. Action of `$uia_alloc/2`.

`$uia_pokes/3` is called in the folloiwng form:

```
'$uia_pokes'(UIABuf,Offset,Insert)
```

`UIA Buf and Offset` are as above. `Insert` is an atom or another UIA. Like `$uia_pokeb/3`, `$uia_pokes/3` views the buffer as a vector of bytes with offset zero specifying the first byte. But instead of replacing just a single byte, `$uia_pokes/3` replaces the portion of the buffer beginning at `Offset` and having length equal to the length of `Insert`, using the characters of `Insert` for the replacement. If `Insert` would extend beyond the end of the buffer, `Insert` is truncated at the end of the buffer. This is illustrated in Figure 6 (*Action of*

1. These procedures can be used to modify system atoms (file names and strings that are represented as UIAs). However, this use is strongly discouraged.

$uia_pokes/3.) .

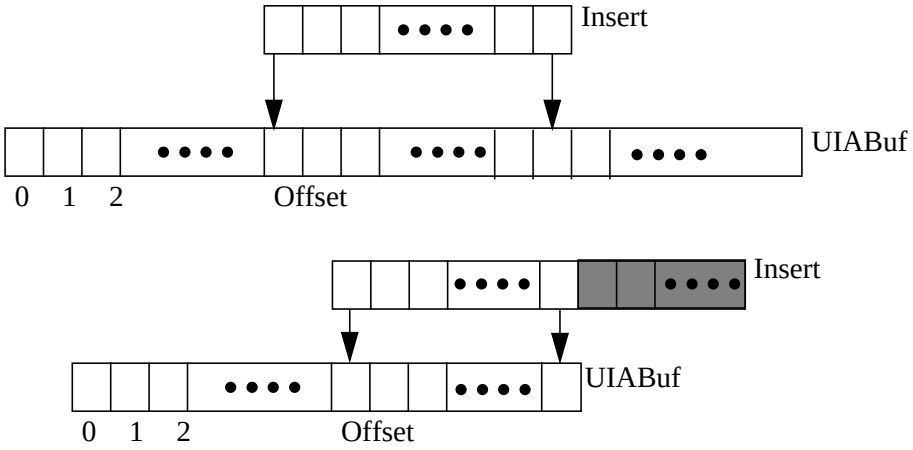


Figure 6. Action of `$uia_pokes/3`.

Accessing UIA Components

$uia_peekb/3, `$uia_peeks/3`, and `$uia_peeks/4` are used to obtain specific bytes and symbols (UIAs) from a buffer created by `$uia_alloc/2`. The parameters for these procedures are specified as follows:

```
'$uia_peekb'(UIABuf,Offset,Value)
'$uia_peeks'(UIABuf,Offset,Extract)
'$uia_peeks'(UIABuf,Offset,Size,Extract)
```

These parameters are interpreted in the same manner as the parameters for `$uia_pokeb/3` and `$uia_pokes/3`, where `Size` must also be an integer. `$uia_peekb/3` binds `Value` to the byte at position `Offset`. `$uia_peeks/3` binds `Extract` to a UIA consisting of the characters beginning at position `Offset` and extending to the end of the buffer. `$uia_peeks/3` binds `Extract` to a UIA consisting of the characters beginning at position `Offset` and extending to position `End` where `End = Offset + Size`. If End would occur beyond the end of the buffer, `Extract` simply extends to the end of the buffer.

An Example.

The following example procedure illustrates how to create a buffer and fill it with the name of a given atom with using $uia_pokes/3.

```
copy_atom_to_uia(Atom, UIABuf) :-
  name(Atom,ExplodedAtom),
  copy_list_to_uia(ExplodedAtom,UIABuf).

copy_list_to_uia(Ints,UIABuf) :-
  length([_|Ints], BufLen),
  '$uia_alloc'(BufLen, UIABuf),
  copy_list_to_uia(Ints, 0, UIABuf).

copy_list_to_uia([],_,_) :- !.

copy_list_to_uia([H | T], N, Buf) :-
  '$uia_pokeb'(Buf,N,H),
  NN is N+1,
  copy_list_to_uia(T, NN, Buf).
```

Below is a full list of the routines which may be used to modify and access component values of a UIA. Details can be found in the ALS Prolog Reference Manual.

$uia_clip/2	- clip the given UIA
$uia_pokeb/3	- modifies the specified byte of a UIA
$uia_peekb/3	- returns the specified byte of a UIA
$uia_pokew/3	- modifies the specified word of a UIA
$uia_peekw/3	- returns the specified word of a UIA
$uia_pokel/3	- modifies the specified long word of a UIA
$uia_peekl/3	- returns the specified long word of a UIA
$uia_poked/3	- modifies the specified double of a UIA
$uia_peekd/3	- returns the specified double of a UIA
$uia_pokes/3	- modifies the specified substring of a UIA
$uia_peeks/3	- returns the specified substring of a UIA
$uia_peeks/4	- returns the specified substring of a UIA
$uia_peek/4	- returns the specified region of a UIA
$uia_poke/4	- modifies the specified region of a UIA

There are two useful routines for dealing with the sizes of UIAs. The call

```
'$uia_size'(UIABuf,Size)
```

returns the actual size (in bytes) of the given UIA. If `Size` is less than or equal to the actual size of the given `UIABuf`, the call

```
'$uia_clip'(UIABuf,Size)
```

reduces the size of `UIABuf` by removing all but one of the trailing zeros (null bytes). When `Atom` is a Prolog atom (symbol or UIA),

```
'$strlen(Atom,Size)'
```

returns the length of the print name of that atom (thus not counting the terminating null byte).

12.4 Observations on Using UIAs.

As indicated by the rules presented in Section 12.2 (*Interning UIAs*) , ALS Prolog automatically handles much of the use of UIAs. The preceeding Section presented predicates for explicitly creating and manipulating UIAs. The routines for explicit manipuulation of UIAs allow one to treat the bytes making up the UIA as raw memory to be manipulated at a low level. Two areas where explicit manipulation of UIAs can be useful are:

- Creating and manipulating data structures not supported by ALS Prolog.
- Communicating with external programs.

One example which in essence combines both uses is communication with external C programs, such as X Windows and Motif, which require both C strings and C structs are function arguments. An analysis of any such situation usually leads to the following observations:

- Strings and structs which are created on the C side of the interface should usually stay there, and pointers to them be passed to the Prolog side.
- Strings and structs which are created on the Prolog side and which are ephemeral in the sense that the C side will consume them when they are initially passed, and no further reference will be made to them from the C side, can be created as UIAs. Note that if after control returns to Prolog, a gar-

bage collection will sooner or later take place. In all likelihood, the UIA object will either pass out of existence, or at least move its location on the heap, so that it C 'holds on' to the pointer it was passed, this pointer will no longer be valid. Thus, one only wants to pass UIAs to C when they are ephemeral in the sense above: C will not hold on to a pointer to the UIA after control returns to Prolog.

- When non-ephemeral objects are to be created under Prolog control, it is best to create these in 'C space' by calling `malloc`. This can be done either by creating a specific C-defined Prolog predicate which carries out the work, or by using the C interface utilities which allow Prolog to call `malloc` and manipulate the allocated C memory.

13 Global Variables, Destructive Update & Hash Tables

ALS Prolog provides a method of globally associating values with arbitrary term (which occur on the heap).. The associations are immune to backtracking. That is, one an association is installed, backtracking to a point prior to creation of the association does not undo the association. (However, see the discussion below for fine points concerning this.) Because both the associated term and value may occur on the heap, both a term and its associated value can contain uninstatiated variables.

13.1 'Named' Global Variables

The underlying primitive predicates set_global/2 and get_global/2 defined in the next section maintain a uniform global association list. This has the disadvantage that as the number of distint associations to be mainted grows, the performance of both set_global/2 and get_global/2 will degrade. The facility described in this section avoids this problem by providing individual global variables which are accessed by programmer-specified unary predicates; hence this mechanism is said to provide 'named global variables.'

make_gv/1
make_gv(Name)
make_gv(+)

This predicate creates a single (primitive) global variable (see the next section), together with predicates for setting and retrieving its value. `If Name` is either an atom or a Prolog string (list of ASCII codes), the call

```
make_gv(Name)
```

allocates a primitive global variable and dynamically defines (asserts clauses for) two predicates, `setNAME/1` and `getNAME/1`, where `NAME` is the atom `Name` or the atom corresponding to the string `Name`. The definitions are installed in the module in which `make_gv/1` is called. These two predicates are used, respectively, to set or get the values of the global variable which was allocated. Here are some examples:

```
?-make_gv('_flag').
yes.

?-set_flag(hithere).
yes.

?-get_flag(X).
X = hithere.

?-make_gv('CommonCenter').
yes.

?-setCommonCenter(travel_now).
yes.

?-getCommonCenter(X).
X = travel_now.
```

13.2 The Primitive Global Variable Mechanism.

The underlying or primitive global variable mechanism is best described in terms of a simple implementation point of view. Global variables are value cells with the following properties:

- They can contain pointers into the heap (but not into the stack)
- These value cells do not lie on either the heap or the stack.
- The pointers contained in these value cells are not affected by either the backtracking process or the garbage collection process.

Figure 7 suggests the global variable mechnism.

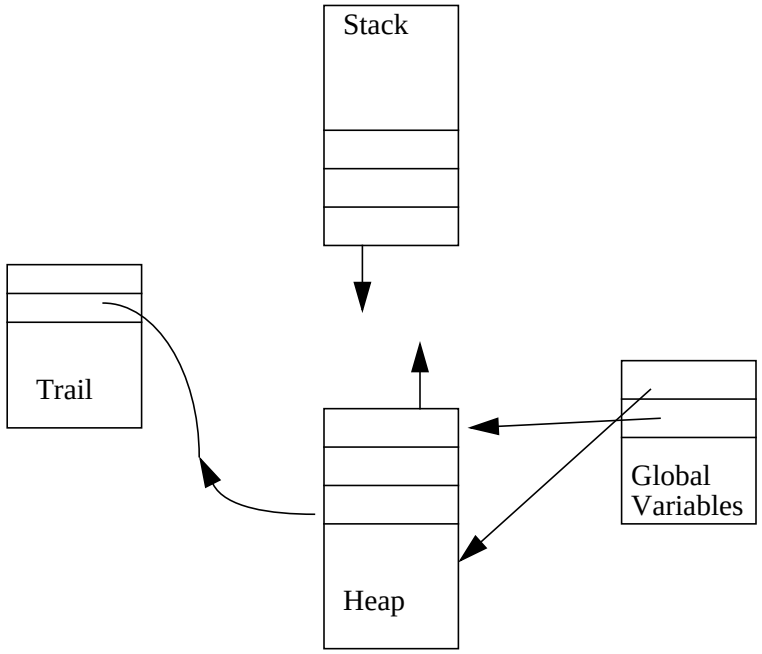


Figure 7. The Global Variables Area and the Heap.

The underlying mechanism is implemented by the following routines:

gv_alloc/1
gv_alloc(Num) - *allocates a global variable*
gv_alloc(+)

gv_free/1
gv_free(Num) - *frees a global variable*
gv_free(+)

gv_get/2
gv_get(Num, Value) - *gets the value of a global variable*
gv_get(+, -)

gv_set/2
gv_set(Num, Value) - *sets the value of a global variable*
gv_set(+, +)

These four predicates implement the primitive global variable mechanism. They achieve an effect often implemented using assertions in the database. The value of the present mechanism is its greater speed, its separation from the database, and its ability to deal with terms from the heap which may incorporate uninstatioated variables. Global variables are referred to by unique identifying integers sequentially starting from 1. The number of available global variables is implementation dependent. Note that the system itself allocates a number of global variables.

`gv_alloc(Num)` allocates a free global variable and unifies the number of this variable with `Num`. `gv_free(Num)` deallocates global variable number `Num`, which can then be reused by subsequent calls to `gv_alloc`/1. Since several global variables are used by the system itself, the first call to `gv_alloc(Num)` normally returns an integer greater than 1.

`gv_set(Num,Value)` sets the value of global variable number `Num` to be Value, which can be any Prolog term, including partially instantiated terms[1]. Correspondingly, `gv_get(Num, Value)` unifies `Value` with the current value of global variable number `Num`. A call to `gv_get(Num, Value)` before a call to `gv_set(Num, Value)` returns the default value for global variables, which is `0`.

Attempts to use `gv_set(Num,Value)` or `gv_get(Num,Value)` without a preceeding call to `gv_alloc(Num)` returning a value for the variable `Num` is an error which will generally cause unpredictable behavior, including system crashes.

The ***immediate*** values of global variables survive backtracking and persist across top level queries. However, if a global variable is set to a structure containing an unbound variable, say X, which is later bound during a computation, the binding of X is an ordinary Prolog binding which will not survive either backtracking or return

1. The following earlier restriction has been removed: “However, it must not be a single free-standing uninstantiated variable. Unpredictable behavior will result if Value is an uninstantiated variable.”

to the top level of the Prolog shell. Thus variables in a structure which is bound to a global variable do not inherit the globalness of the outermost binding.

Here are some examples:

```
?- gv_alloc(N), gv_set(N,hi), write(hi).
hi
N = 2
yes.

?- gv_get(2,V),write(V).
hi
V = hi
yes.

?- gv_set(2,bye).
yes.

?- gv_get(2,V1),write(V1),nl,fail;
        gv_get(2,V2),write(V2).
bye
bye
V1 = _4
V2 = bye
yes.

?- gv_get(2,V).
V = bye
```

Note that `gv_set/2` is a constant time operation so long as the second argument is an atom or integer. Otherwise, it requires time linearly proportional to the current depth of the choicepoint stack.

13.3 Destructive Modification/Update of Compound Terms

ALS Prolog provides a predicate which allows programs to destructively modify arguments of compound terms (or structures). This predicate is `mangle/3`. The

effects of mangle/3 are destructive in the sense that they survive backtracking. The calling pattern for this predicate is similar to `arg/3:`

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

This call destructively modifies an argument of the compound term `Structure` in a spirit similar to Lisp's `rplaca` and `rplacd`. `Structure` must be instantiated to a compound term with at least `N` arguments. The `Nth` argument of `Structure` will become `NewArg`. Lists are considered to be structures of arity two. `NewArg` must satisfy the restriction that `NewArg` is not itself an uninstatiated variable (though it can be a compound term containing uninstatiated variables). Modifications made to a structure by `mangle/3` will survive failure and backtracking.

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

Here are some examples:

```
?- Victim = doNot(fold,staple,mutilate),
       mangle(2,Victim,spindle).
Victim = doNot(fold,spindle,mutilate)
yes.
```

13.4 'Named' Hash Tables

The allocation and use of hash tables is supported by exploiting the fact that the implementation of terms is such that a term is an array of (pointers to) its arguments. So hash tables are created by combining a term (created on the heap) together with access routines implemented using basic hashing techniques. The destructive update feature mangle/3 is used in an essential manner. As was the case with global variables, at bottom lies a primitive collection of mechanisms, over which is a more easily usable layer providing 'named' hash tables.

The predicate for creating named hash tables is

make_hash_table/1
make_hash_table(Name) - ***creates a hash table with access predicates***
make_hash_table(+)

If Name is any atom, including a quoted atom, the goal make_hash_table(Name) will create a hash table together a set of access methods for that table. The atom `Name` will be used as the suffix to the names of all the hash table access methods. Suppose for the sake of the following discussion that `Name` is bound to the atom '`_xamp_tbl`'. Then the goal

```
make_hash_table('_xamp_tbl')
```

will create the following access predicates:

`reset_xamp_tbl` - throw away old hash table associated with the '_xamp_tbl' hash table and create a brand new one.

`set_xamp_tbl(Key,Value`

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

`get_xamp_tbl(Key,Value)`

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

del_xamp_tbl(Key,Value)

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

`pget_xamp_tbl(KeyPattern,ValPattern)`

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

`pdel_xamp_tbl(KeyPattern,ValPattern)`

– This functions the same as `pget_xamp_tbl` except that

the association is deleted from the table once it is retrieved.

Consider the following example (where we have omitted all of the ‘yes’ replies, but retained the ‘no’ replies):

```
?- make_hash_table(‘_assoc’).
?- set_assoc(a, f(1)).
?- set_assoc(b, f(2)).
?- set_assoc(c, f(3)).

?- get_assoc(X, Y).
no.

?- get_assoc(c, Y).
Y = f(3)

?- pget_assoc(X, Y).
X = c
Y = f(3);

X = b
Y = f(2);

X = a
Y = f(1);

no.

?- del_assoc(b, Y).
Y = f(2)
?- pdel_assoc(X, f(3)).
X = c

?- pget_assoc(X, Y).
X = a
Y = f(1);
```

```
no.

?- reset_assoc.
yes.

?- pget_assoc(X,Y).

no.
```

13.5 Primitive Hash Table Predicates

The core hash tables are physically simply terms of the form

```
hashArray(.........)
```

We are exploiting the fact that the implementation of terms is such that a term is an array of (pointers to) its arguments. So what makes a hash table a hash table below is the access routines implemented using basic hashing techniques. We also exploit the destructive update feature `mangle/3.` Each argument (entry) in a hash table here is a (pointer) to a list `[E1, E2, ....]` where each `Ei` is a cons term of the form

```
[Key Value]
```

So a bucket looks like:

```
[ [Key1 Val1], [Key2 Val2], ....]
```

where each `Keyi` hashes into the index (argument number) of this bucket in the term

```
hashArray(.........)
```

The complete hash tables are terms of the form

```
hastTable(Depth,Size,RehashCount,hashArray(....))
```

where:

Depth = the hashing depth of keys going in;

Size = arity of the `hashArray(...)` term;

RehashCount = counts (down) the number of hash entries which have been made; when then counter reaches 0, the table is expanded and rehashed.

The basic (non-multi) versions of these predicates overwrite existing key values; i.e., if `Key-Value0` is already present in the table, then hash inserting `Key-Value1` will cause the physical entry for `Value0` to be physcially altered to become `Value1` (using `mangle/3`).

The "-multi" versions of these predicates do NOT overwrite existing values, but instead treat the `Key-___` cons items as tagged pushdown lists, so that if

```
[Key Value0]
```

was present, then after hash_multi_inserting `Key-Value1`, the `Key` part of the bucket looks like: `[Key [Value1 Value0] ];` i.e., it is

```
[Key, Value1 Value0]
```

Key hashing is performed by the predicate

```
hashN(Key,Size,Depth,Index).
```

14 Freeze, Exceptions, Events, Interrupts, Signals.

14.1 Freeze

ALS Prolog supports a ‘freeze’ control construct similar to those that appear in some other prolog systems. Using ‘freeze’, one can implement a variety of approaches to co-routining and delayed evaluation.

freeze/2
freeze(Var, Goal)
freeze(?, +)

In normal usage, `Var` is an uninstantiated variable which occurs in Goal. When invoked in module `M`, the call

```
freeze(Var, Goal)
```

behaves as follows:

1. If `Var` is instantiated, then `M:Goal` is executed;
2. If `Var` is not instantiated, then the goal `freeze(Var, Goal)` immediately succeeds, but creates a ‘delay term’ (on the heap[1]) which encodes information about this goal. If `Var` becomes instantiated (at some point) in the future, at that time, the goal `M:Goal` is run (with, of course, `Var` instantiated).

For example, here is an example of an extremely simple producer-consumer coroutine:

```
pc2 :-
    freeze(S, produce2(0,S)), consume2(S).

produce2(N, [N | T])
    :-
    M is N+1,
    write(‘-p-’),
```

1. See [carlsson] for general information on delay terms and implmentation strategies.

```
    freeze(T, produce2(M,T)).

consume2([N | T])
    :-
    write(n=N),nl,
    ((N > 3, 0 is N mod 3) -> gc; true),
    (N < 300 ->
        consume2(T) ; true).
```

Without the presence of the ‘freeze’ constructs, this program will simply loop in produce2/2, doing nothing but incrementing the counter and printing ‘-p-’ on the terminal. However, using the freeze construct, the program ‘alternates’ between produce2/2 and consume2/1, producing the following behavior on the terminal:

```
?- pc2.
-p-n = 0
-p-n = 1
-p-n = 2
-p-n = 3
-p-n = 4
-p-n = 5
<...snip...>
-p-n = 294
-p-n = 295
-p-n = 296
-p-n = 297
-p-n = 298
-p-n = 299
-p-n = 300

yes.
?-
```

Here is one very simple illustrative example:

```
u :-
    freeze(W1, silly(W1,yellow)),
```

```
    freeze(W2, grump(W2,blue)),
    W2=W1,
    W2 = igloo.

grump(A,B)
    :-
    write(grump_running(A,B)),nl,flush_output.

silly(A,B)
    :-
    write(silly_running(A,B)),nl,flush_output.
```

Running u/0 yields:

```
?- u.
silly_running(igloo,yellow)
grump_running(igloo,blue)

yes.
```

Uisng silly and grump from above, here is another example:

```
u1 :-
    freeze(W1, silly(W1,yellow)),
    u11(W1).

u11(W1)
    :-
    freeze(W2, grump(W2,blue)),
    W2=W1,
    u111(W2).

u111(W2)
    :-
    freeze(W3, grump(W3,purple)),
    W3 = W2,
    u1_4(W3).
```

```
u1_4(W3)
    :-
    W3 = igloo.

u11(W1).
u111(W1).
u1_4(W3).
```

Runing this produces:

```
?- u1.
silly_running(igloo,yellow)
grump_running(igloo,blue)
grump_running(igloo,purple)

yes.
```

The following example[1] illustrates the interaction of freeze with backtracking:

```
fred(2) :- write(fred(2)),nl.
fred(3) :- write(fred(3)),nl.
fred(4) :- write(fred(4)),nl.

freeze_backtrack
    :-
    freeze(X, write(thaw(X))), fred(X), fail.
```

The output here is:

```
?- freeze_backtrack.
thaw(2)fred(2)
thaw(3)fred(3)
thaw(4)fred(4)

no.
```

Finally, here is an example[2] of cascading freezes:

1. Due to Bill Older.

```
fd([], 1).
fd([A | As], B)
    :-!,
    freeze(A, fd(As, B)).

fdtest([A,B,C,D]) :-
    fd([A], B),
    fd([A,B], C),
    fd([B,C], D).
```

Here are two different uses of fdtest:

```
?- fdtest([A,B,C,D]).

A-> fd([],B),user:fd([B],C)
B-> fd([C],D)
C = C
D = D

yes.
?- fdtest([A,B,C,D]), A = 5.

A = 5
B = 1
C = 1
D = 1

yes.
?-
```

14.2 Exceptions.

The exception mechanism of ALS Prolog allows programs to react to extraordinary circumstances in an efficient and appropriate manner. The most common extraordinary circumstance to be dealt with is errors. Often an error (perhaps inappropriate

2. Due to Bill Older.

user input, etc.) is detected deep in the calling sequence of predicates in a program. The most appropriate reaction on the part of the program may be to return to a much earlier state. However, if the code is written to support such a return using the ordinary predicate calling mechanisms, the result is often difficult to understand and has poor effeiciency. The exception mechanism allows the program to mark a point in its calling state, and to later be able to return directly to this marked point independently of the pending calls between the marked state and the later state. This notion is illustrated in Figure 8

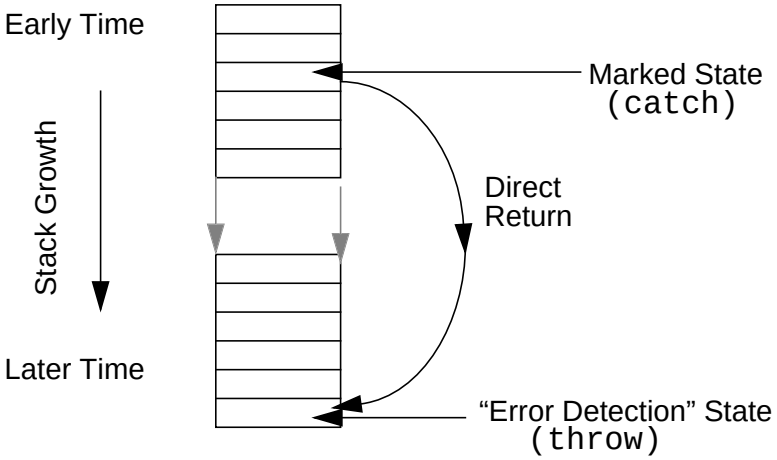


Figure 8. Direct Return to an Earlier State.

This exception mechanism is implemented using two predicates, catch/3 and throw/1 .

catch/3
catch(WatchedGoal, Pattern, ExceptionGoal)
catch(+, +, +)

throw/1
throw(Term)
throw(+)

The two predicates `catch/3` and `throw/1` provide a more sophisticated con-

trolled abort mechanism than the primitive builtins `catch/2` and `throw/0` (although the former are implemented in terms of the latter, which are described below). `catch/3` is used to mark a state in the sense of the discussion above. We will say that the call

```
catch(WatchedGoal, Pattern, ExceptionGoal)
```

catches a term `T` if `T` unifies with `Pattern`. A call `throw(T)` is ***caught by*** a call on `catch/3` if the argument `T` is caught by the call on `catch/3`, an there is no call on `catch/3` between the given calls on `catch/3` and `throw/1` which also catces `T`.

If `M` is the current module for the call on `catch/3,` then the first argument of `catch/3, WatchedGoal`, is run in module `M` just as if by `call/1;` i.e., as if `catch/3` were `call/1.` If there is no subsequent call to `throw/1` which is uncaught by an intervening call to `catch/3,` then the call on `catch/3` is exactly like a call on `call/1.` However, if i) there is a subsequent call to `throw/1` whose argument `Ball` unifies with the second arguement of the call the `catch/3,` and if ii) there is no interposed call to `catch/3` whose second argument also unifies with `Ball`, then all computation of the call on the first argumentt, `WatchedGoal`, is aborted, and the third argument of the call to `catch/3, ExceptionGoal`, is run as a result of the call to `throw/1`.

When the system executes throw/1, it will behave as if the head of throw/1 failed. However, instead of backtracking to the most recent choicepoint, the system will instead backtrack to the state it was in just before the most recent enclosing catch/3 whose second argument unifies with the argument of the throw/1 call; then the system will run the corresponding ExceptionGoal. catch and throw are dynamically scoped in that throw/1 must be called somewhere in an execution of WatchedGoal which is initially invoked by catch/3. If throw/1 is called outside the scope of some invocation of catch/3 (meaning, with nothing to catch its abort of execution), the system aborts to the Prolog shell. Figure 9 below illustrates the behavior of these

predicates.

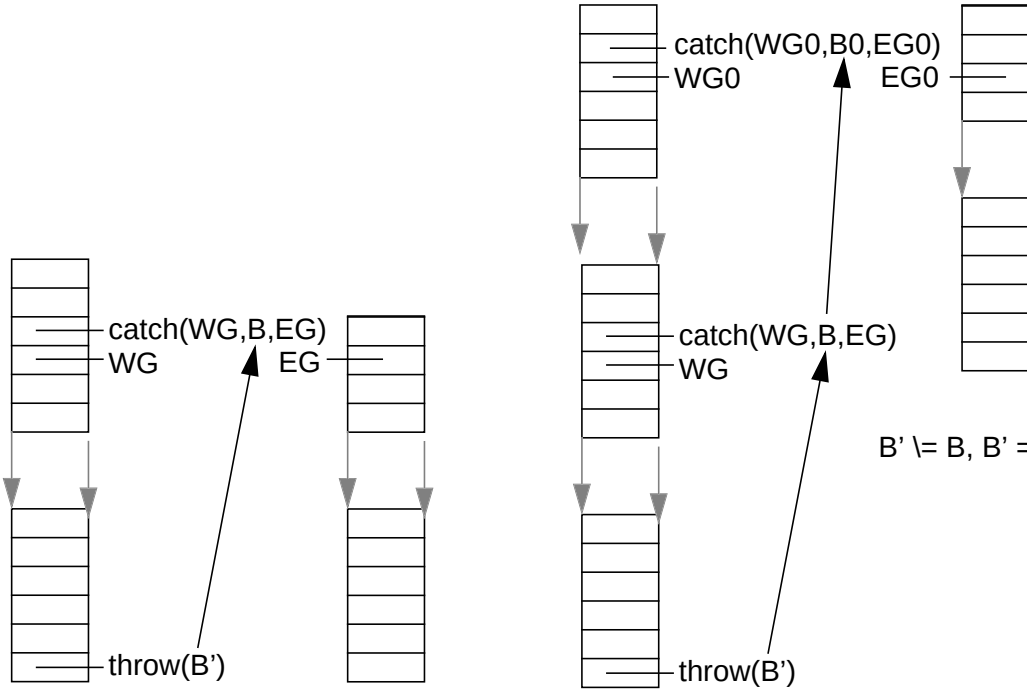


Figure 9. Action of `catch/3` and `throw/1`.

Consider the sample code:

```
ct :-          catch(c1, p1(X), e(c1,X)).
c1 :-        write(c1),nl, catch(c2, p2(X), e(c2,X)).
c2 :-        write(c2),nl, catch(c3, p1(X), e(c3,X)).
c3 :-          write('c3-->'), read(Item),
               write(throwing(Item)),nl, throw(Item).
e(H,I) :-      printf("Handler %t caught item
  %t\n",[H,I]).
```

This leads to the following execution behavior on a TTY interface, with the user input indicated in Helvetica.

```
?- ct.
```

```
c1
c2
c3-->p1(a).
throwing(p1(a))
Handler c3 caught item a
yes.
?- ct.
c1
c2
c3-->p2(a).
throwing(p2(a))
Handler c2 caught item a
```

The file *catch.pro* records the item being thrown by `throw/1` in the Prolog database, while the file *catchg.pro* records the item in a global variable. For small items, there is no significant difference between the two versions, but for large items, the *catchg* version will be more efficient.

catch/2
catch(WatchedGoal,ExceptionGoal)
catch(WatchedGoal,ExceptionGoal)

throw/0
throw

The two predicates `catch/2` and `throw/0` provide the primitive form of controlled abort underlying `catch/3` and `throw/1`. The builtin `abort/0` normally aborts to the Prolog shell.

- `WatchedGoal` is simply run from the current module as if by `call/1`.
- `ExceptionGoal` is a goal that will be run as a result of a call to `throw/0` during the execution of the `WatchedGoal`.

When the system executes `throw/0` it will behave as if the head of `throw/0` failed. However, instead of backtracking to the most recent choicepoint, the system will instead backtrack to the state it was in just before the most recent enclosing `catch/2` and then run the corresponding `ExceptionGoal`. `catch` and `throw`

are dynamically scoped in that `throw/0` must be called somewhere in the execution of `WatchedGoal` which is initially invoked by `catch/2`. If `throw/0` is called outside the scope of some invocation of `catch/2` (meaning, with nothing to catch its execution), the system aborts to the Prolog shell.

```
?- throw.
Execution aborted.
If we define the following clauses:
goal(Person) :-
       printf("%t: Chris, take out that
  garbage!\n",[Person]),
       responseTo(Person), okay(Person).

responseTo('Mom') :- throw.
responseTo('Dad').

okay(Person) :-
       printf("Chris: Okay %t, I'll do
  it.\n",[Person]).

interrupt :- printf("Chris: I want to go hiking with
  Kev.\n").
Then
?- catch(goal('Mom'),interrupt).
Mom: Chris, take out that garbage!
Chris: I want to go hiking with Kev.
yes.

?- catch(goal('Dad'),interrupt).
Dad: Chris, take out that garbage!
Chris: Okay Dad, I'll do it.
yes.
```

Notice that with Mom, it's okay to interrupt, but you don't try it with Dad. In the above example, Chris responds to Dad with the `okay/1` predicate, but Chris did not respond to Mom because the `okay/1` predicate was never reached. After a

`throw/0` predicate is executed, control is given to the `ExceptionGoal`. After the `ExceptionGoal` is run, control starts after the call to `catch/2` which handled the exception. Invocations of catch/2 may be nested and a throw/0 will always go to the most recent enclosing catch/2.

14.3 Interrupts.

Each time a procedure is called, ALS Prolog compares the distance between the top of the heap and its boundary to see if a garbage collection is necessary. If this distance ever becomes less than a pre-defined value, called the *heap safety value*, the program is interrupted, and the garbage collector is invoked. The test is done at the entrance to every procedure. This check is the basis for the internal ALS Prolog interrupt mechanism.

When a procedure is called by a WAM call or execute instruction (cf. [warren83]), control is transfered to a location in the name table entry for the procedure which is being called. The first action carried out in this patch of code is the heap overflow check. If no overflow has been detected, control continues on. This overflow check is useful as a general interrupt mechanism in Prolog. Since it is always carried out upon procedure entry, and since all calls must go through the procedure table, any call can be interrupted. If the overflow check believes that the heap safety value is larger than the current distance between the heap and backtrack stack, the next call will be stopped. The key word is ‘ believes’: the heap safety value could have been set to an absurdly large value leading to the interruption.

The key to using this as the basis for a general Prolog interrupt mechanism is to provide a method of specifying the reason for the interrupt, together with an interrupt handler which can determine the reason for the interrupt and dispatch accordingly. The entire mechanism is implemmented in Prolog code. Either ALS Prolog system code or user code can trigger an interrupt by setting the heap safety to a value which guarantees that the next call will be interrupted. When the interrupt occurs, the interrupted call is packaged up in a term and passed as an argument to the interrupt handler. The continuation pointer for the handler will point into the interrupted clause, and the computation will continue where it would have continued if the call had never been interrupted after the handler returns.

An example will help make this clearer. Suppose the goal

```
:- b, c, f(s,d), d, f.
```

is running , that the call for `c/0` has returned, and that `f/2` is about to be called. Something happens to trigger an interrupt (i.e., to set the heap safety value to a very large value) and `f/2` is called. The heap overflow check code will run and the goal will now operationally look as though it were

```
:- b, c, '$int'(f(s,d)), d, f.
```

Rather than `f/2` running, `$int/1` will run. When `$int/`1 returns, `d/0` will run, which is what would have happened if `f/2` had run and returned. If `$int/1` decides to run `f/2`, all it has to do is call `f/2`. `$int/1` can leave choice points on the stack, and also be cut, since it is exactly like any other procedure call. Any cuts inside `$int/1` will have no effects outside of the call. In other words, it is a fairly safe operation. Once the interrupt handler is called, the interrupt trigger should be reset, or the interrupt handler will interrupt itself, and go into an infinite loop.

If some thought is given to the possibilities of this interrupt machanism, it becomes apparent that it can be used for a variety of purposes, as sketched below.

A clause decompiler in Prolog itself: The `$int/1` code might be of the form

```
'$int'(Goal) :-
   save Goal somewhere,
   set a 'decompiler' interrupt.
```

The goal would be saved somewhere, and the interrupt code would merely return after making sure that the next call would be interrupted. It is not necessary to call Goal, because nothing below the clause being decompiled is of interest.

A debugging trace mechanism: The `$int/1` code would be of the form

```
'$int'(Goal) :-
   show user Goal,
   set a 'trace' interrupt and call Goal.
```

Here, the code will show the user the goal and then call it, after making sure that all subgoals in Goal will be interrupted.

A `^C` trapper: The `$int/1` code would keep the current goal pending. Then the user could be given a choice of turning on the trace mechanism, calling a break package which would continue the original computation when it returned, or even

stop the computation altogether.

In order for the above operations to take place, the interrupt handler needs to know which interrupt has been issued. This is done through the ***magic value***. `Magic` is a global variable, which is provided as the first argument to the `$int/2` call

```
'$int'(Magic,Goal)
```

Calling `$int/2` with the value of `Magic` passed (as first argument) means that the proper interrupt handler will be called. If the value of `Magic` is allowed to be a regular term, information can be passed back from an interrupt, such as the accumlated goals from a clause which is being decompiled. The mechanisms of setting interrupts clearly must include the setting of `Magic` to appropriate values.

In order to write code such as the decompiler in Prolog, several routines are needed. The system programmer must be able to set and examine the value of `Magic`. This is done with the

```
setPrologInterrupt/1
getPrologInterrupt/1
```

calls. The programmer must also be able to interrupt the next call. This is done with the

forcePrologInterrupt/0

call. For example, the goal

```
:- forcePrologInterrupt, a.
```

will call

```
'$int'(Magic,a)
```

If the clause

```
b :- forcePrologInterrupt.
```

is called by the goal

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

then

```
'$int'(Magic,a)
```

will be called once again, since after `b` returns, `a/0` is the next goal called. Finally, there must be a way of calling a goal without interrupting it, but setting the interrupt so that the the goal after the goal called will be interrupted. If `a/0` is to be called and the next call following it is to be interrupted, the call

```
:- callWithDelayedInterrupt(a)
```

is used. For example, if `a/0` is defined by

```
a :- b.
```

then

```
:- callWithDelayedInterrupt(a).
```

will call

```
:- '$int'(Magic,b).
```

not

```
:- '$int'(Magic,a).
```

However, if `a/0` is merely the fact

```
a.
```

then the call

```
:- callWithDelayedInterrupt(a),b.
```

will end up calling

```
:- '$int(Magic,b).
```

As an extended example of the use of these routines, we will construct a simple clause decompiler. The code sketched earlier outlines the general idea:

```
'$int'(Goal) :-
   save Goal somewhere,
   set a 'decompiler' interrupt.
```

The goal can be saved for the next call inside a term in the variable `Magic`. The clause so far would be

```
'$int'(s(Goals,Final),NewGoal) :-
   setPrologInterrupt(s([Goal|Goals],Final)),
```

```
      forcePrologInterrupt.
```

The term being built inside `Magic` has the new goal added to it, and the trigger is set for the next call. To start the decompiler, the clause should be called as though it were to be run. However, each subgoal will be interrupted and discarded before it can be run. The starting clause would be something of the form:

```
$source(Head,Body) :-
   setPrologInterrupt(s([],Body)),
   callWithDelayedInterrupt(Head).
```

First, the value of `Magic` is set to the decompiler interrupt term with an initially empty body and a variable in which to return the completed body of the decompiled clause. The goal is then called with `callWithDelayedInterrupt/1`, which will make sure that the next goal called after `Head` will be interrupted. The above clause for `$int/2` will then catch all subgoals. The head code for `Head` will bind any variables in `Head` from values in the head of the clause, and all variables that are in both the head and body of the clause will be correct in the decompiled clause, since an environment has been created for the clause. Since the clause is actually running, each subgoal will pick up its variables from the clause environment. If the decompiler should ever backtrack, the procedure for `Head` will backtrack, going on to the next clause, which will be treated in the same way. The only tricky thing is the stopping of the decompiler. The two clauses given above will decompile the entire computation, including the code which called the decompiler. The best method is to have a goal which the decompiler recognizes as being an ‘end of clause flag’. However, having a special goal which would always stop the decompiler would mean that the decompiler would not be able to decompile itself. So some way must be found to make only the particular call to this ‘distinguished’ goal be the one at which the decompiler will stop. The clauses above can be changed to the following to achieve this goal:

```
‘$int’(s(ForReal,Goals,Final),$endSource(Variable):-
   ForReal == Variable,!.

‘$int’(s(ForReal,Goals,Final),Goal) :-
   setPrologInterrupt(s(ForReal,
                        [Goal|Goals],Final)),
```

```
      forcePrologInterrupt.

   '$source'(Head,Body) :-
      setPrologInterrupt(s(ForReal,[],Body)),
      callWithDelayedInterrupt(Head),
      '$endSource'(ForReal).
```

Here, `$source/2` has a variable `ForReal` in its environment. This is carried through the interrupts inside the term in `Magic`. If the interrupted goal is ever `$endSource(ForReal)`, the decompiler stops. Note that `$endSource(ForReal)` will be caught when `$source/2` is called, since all subgoals are then being caught, and it's argument will come from the environment of `$source/2`. Otherwise, the interrupted goal is added to the growing list of subgoals, and the computation continues. If `$source/2` is called by `$source/2`, there will be a new environment and the first `$endSource/1` encountered will be caught and stored, but not the second one. The complete code for the decompiler appears in the file *builtins.pro.* This decompiler is used as the basis for `listing` as well as for `retract`. In addition, it is used to implement the debugger, found in the file *debugger.pro.*

14.4 Events

The event handling mechanism[1] provides implements both the system-level error and exception mechanisms, together with the general user-level event mechanisms such as coupling to signals and application-based interrupts. The event mechanism in ALS Prolog is based on the design presented in "Event handlin in prolog", by Micha Meier, in E.Lusk & R.Overbeek (eds), ***Logic Programming, Proceedings of the North American Converence, 1989,*** MIT Press, pp. 871ff. Most of the machinery of the event mechanism is readily discernible in the builtins file *blt_evt.pro.*

At the present time, there are five predefined types of events:

`sigint`	- raised when control-C or its equivalent is hit
`reisscntrl_c`	- raised for "reissued control-C"

1. The particular implementation in ALS Prolog was developed by Kevin Buettner.

`libload` - raised when the stub of a library predicate is encountered

`prolog_error` - raised by prolog errors (mostly from builtins)

undefined_predicate

- raised when an undefined predicate is encountered

Events are handled (and thereby defined) by a local or global event handler. Global handlers are specified in a simple database builtins:global_handler/3:

```
global_handler(EventId, Module, Procedure):

sigint,builtins,default_cntrl_c_handler
reisscntrl_c,builtins,silent_abort
libload,builtins,libload
prolog_error,builtins,prolog_error
undefined_predicate,builtins,undefined_predicate
```

The `global_handler/3` database is best manipulated using the following predicates (which are ***not*** exported from the module `builtins`):

set_event_handler/3
set_event_handler(Module, EventId, Proc)
set_event_handler(+, +, +)

remove_event/1
remove_event(EventId)
remove_event(+)

Additional global event handlers, for example to handle the signal sigalarm , are installed using `set_event_handler/3`, and removed used `remove_event/1.`

An event can be triggered by a program (including system programs) by the following predicate (which ***is*** exported from module builtins):

trigger_event/2
trigger_event(EventId, ModuleAndGoal)
trigger_event(+, +)

The argument ModuleAndGoal is normall of the form `Module:Goal.`

Local event handlers are installed and manipulated using the trap/2 system predicate:

trap/2
trap(Goal,Handler)
trap(Goal,Handler)

Here, `Handler` is a local handler such that

1. `Hander` is in force while `Goal` is running;
2. `Hander` ceases to be in force when `Goal` succeeds or fails;
3. `Hander` returns to force if `Goal` is backtracked into after succeeding
4. `Hander` is capable of dealing with any events which occur while `Goal` is running;

trap/2 is a module closure (meta-predicate), so that the module in which it is called is available to its implementing code. Regarding item 4 above, Handler is usually devoted to one particular type of event, such as handling specific kinds of signals; it will deal with all other events by propagating them to the appropriate "higher level" handlers, either surrounding local handlers, or the global handlers. This propagation is carried out using the following predicate:

propagate_event/3
propagate_event(EventId,Goal,Context)
propagate_event(EventId,Goal,Context)

For example, here is an alarm handler which will be discussed in more detail in the section on signals:

```
alarm_handler(EventId, Goal, Context)
    :-
    EventId \== sigalrm,
    !,
    propagate_event(EventId,Goal,Context).
```

```
alarm_handler(_,Goal,_)
    :-
    write('a_h_Goal'=Goal), nl,
    setSavedGoal(Goal),
    remQueue(NewGoal),
    NewGoal.
```

Note that the first clause uses `propagate_event/3` to pass on all events except `sigalrm`, which is handled by the second clause.

14.5 Signals.

ALS Prolog provides a strong mechanism for interfacing to external operating system signaling mechanisms. The machinery allows the programmer to connect external signals to internal Prolog events as described generally as described in the previous section. The details for the coupling are found in the builtins file *blt_evt.pro.* The connection between signal numbers and signal names is provied by the predicate `signal_name/2:`

```
signal_name(1,sighup).
signal_name(2,sigint).   %% interrupt (as in Cntrl-C)
signal_name(3,sigquit).
.....
signal_name(13,sigpipe).
signal_name(14,sigalrm).    %% alarm clock
signal_name(15,sigterm).
.....
signal_name(29,siglost).
signal_name(30,sigusr1).
signal_name(31,sigusr2).
```

This database is used by the predicate `signal_handler/3` to convert from numeric signal idenfiers to symbolic identifiers:

```
signal_handler(SigNum,Module,Goal)
    :-
    signal_name(SigNum,SigName),
```

```
        !,
        get_context(Context),
        propagate_event(SigName,Module:Goal,Context).
```

`signal_handler/3` is called by the underlying C-defined signal handling mechanism to pass in signals from the operating system. At the present time, there are only two signals for which this mechanism is in place: `sigint (control-C)` and `sigalrm`.

The alarm mechanism is currently only available for Unix-based versions of ALS Prolog. Alarm signals are set using the predicate:

alarm/2
alarm(First, Interval)
alarm(+, +)

Both `First` and `Interval` should be non-negative real numbers. `First` specifies the number of seconds until the first alarm signal is sent to the ALS Prolog process. If `Interval > 0,` an alarm signal is sent to the process every `Interval` seconds after the first signal is sent. Thus `alarm(First, 0)` will result in only one alarm signal (if any) being sent. A subsequent call to `alarm/2` causes the alarm mechanism to be reset according to the parameters of the second call. Thus, even if the first alarm has not yet been sent, a call `alarm(0, 0)` will turn off all alarm signals (until another call to `alarm/2` is used to set the alarms again).

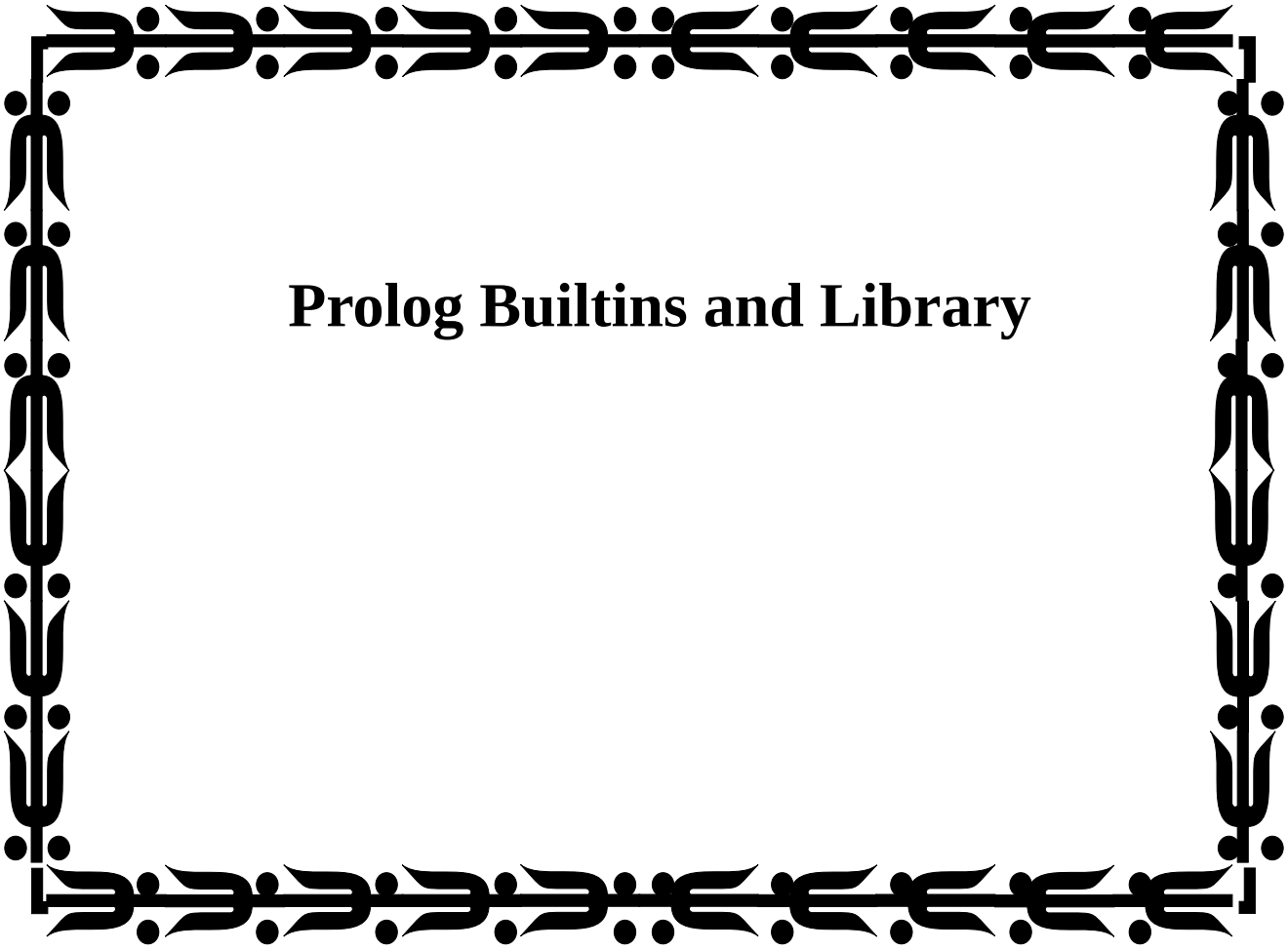
The sample program *par1.pro* illustrates the use of these mechanisms in implementing a simple producer-consumer program.; the entry point is `main/0.`

```
main :-
    trap(main0,alarm_handler).

main0 :-
    initQueue,
    produce(0,L),
    consume(L).

consume(NV) :-
    nonvar(NV),
```

```
    consume0(NV).

consume(NV) :-
    consume(NV).

consume0([H|T]) :-!,
    write(H),nl,
    consume(T).
consume0([]).

produce(100,[]) :- !.
produce(N,[N|T]) :-
    NN is N+1,
    sleep(produce(NN,T)).
/*-----------------------------*
 | Process Management
 *-----------------------------*/
alarm_handler(EventId, Goal, Context) :-
    EventId \== sigalrm,
    !,
    propagate_event(EventId,Goal,Context).

alarm_handler(_,Goal,_) :-
    write('a_h_Goal'=Goal), nl,
    setSavedGoal(Goal),
    remQueue(NewGoal),
    NewGoal.

/*-----------------------------------------------*
 | sleep/1
 |  - put a goal to sleep to wait for the next alarm.
 *-----------------------------------------------*/

:- compiletime, module_closure(sleep,1).
```

```
sleep(M,G) :-
    addQueue(M:G),
    getSavedGoal(SG),
    set_alarm,
    SG.

set_alarm :-
    alarm(1.05,0).
/*-------------------------------------------------*
 | Queue Management:
 |
 |  initQueue/0    -- initializes goal queue to empty
 |  remQueue/1     -- removes an element to the queue
 |  addQueue/1      -- adds an element to the queue
 *-------------------------------------------------*/
initQueue :-
    setGoalQueue(gq([],[])).

remQueue(Item) :-
    getGoalQueue(GQ),
    arg(1,GQ,[Item|QT]),    %% unify Item, QT, and
                            %% test for nonempty
   remQueue(QT,GQ).     %% fix queue so front is gone

    %% Queue is empty:
remQueue([],GQ) :-!,
   mangle(1,GQ,[]),       %% adjust front to be empty
   mangle(2,GQ,[]).   %% adjust rear to be empty also

    %% Queue is not empty:
remQueue(QT,GQ) :-
   mangle(1,GQ,QT).  %% adjust front to point at tail

addQueue(Item) :-
    getGoalQueue(Q),
```

```
    arg(2,Q,Rear),          %% get Rear of Queue
    addQueue(Rear,Q,[Item]).

addQueue([],Q,NewRear) :-!,
    mangle(1,Q,NewRear),
    mangle(2,Q,NewRear).

addQueue(Rear,Q,NewRear) :-
    mangle(2,Rear,NewRear),   %% add the new rear
   mangle(2,Q,NewRear).  %% new rear now rear of queue
```



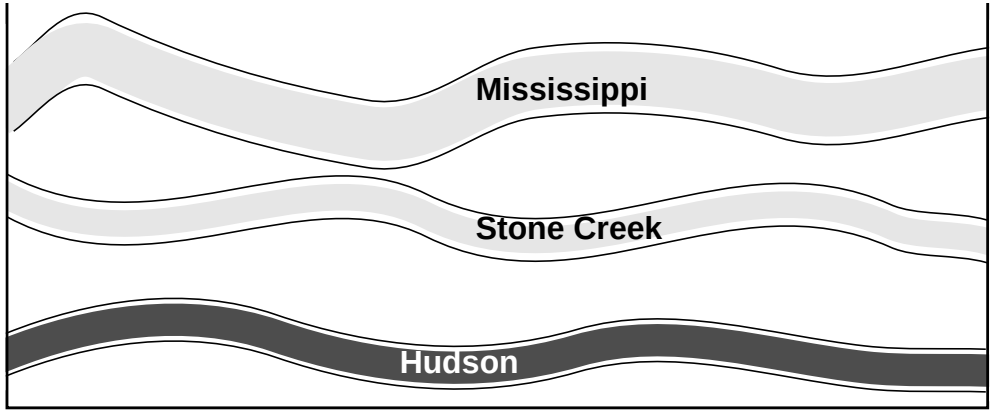
Prolog Builtins and Library

15 Prolog I/O

Most programs need to communicate with the outside world. There are a wide variety of outside entities with which a program can communicate, ranging from the screen, keyboard, and files to other programs over networks. This section describes the facilities which ALS Prolog provides for input/output (I/O) communication. The initial sections describe the fundamental ***stream***-based communication facilities of ALS Prolog. Following this, the earlier so-called ***DEC-10-style*** facilities are described; these are implemented in terms of the stream-based facilities[1].

15.1 Streams, Sources, and Sinks.

No matter how sophisticated the point of view one chooses to take, at bottom, computer communication is based on the notion of ***sequences of characters.*** Such sequences are generated by typing on keyboards, can appear on screens, can be recorded in files, can be transmitted across networks, etc. These sequences of characters are called ***streams***, or, when more precision is necessary, ***character streams***. The word 'stream' is used in this context both because it conveys a sense of motion and because it also conveys a sense of direction, as with the natural notion of stream illustrated in Figure 10 (*Natural Streams.*)



1. Except for the most primitive aspects, the I/O system for ALS Prolog is entirely implemented in ALS Prolog itself. Almost all of this code is contained in the following builtins files: *sio.pro, sio_wt.pro, sio_rt.pro, and blt_io.pro.*

Figure 10. Natural Streams.

In addition, natural streams have a notion of ***source*** (the 'headwaters') and ***sink*** or ultimate destination (the 'estuary'). Finally, a natural stream also conveys the notion of a point on the bank where one can stand and watch the stream flow by. All of these natural notions have corresponding concepts in the computer notion of character streams.

From the logical point of view, the notions of stream, source, and sink are fundamental notions. However, as indicated above, a stream is a finite or potentially infinite sequence of characters. Every stream is associated with a source and a sink. When a stream is manipulated by a program, the program itself is either the source or the sink of the stream. If the program is ***consuming*** the stream, the program is then the ***sink*** for the stream. If the program is ***producing*** or ***generating*** the stream, the program is then the ***source*** of the stream. (It is also possible for the same program to be both source and sink for a stream.)

When the program is the sink for the stream, in general some other entity in the computing environment is the source for the stream. Possible external sources include files, keyboards, devices such as tapes cd-roms, and other programs communicating with the program in question via interprocess communication facilities, either locally or remotely. When the program is the source for the stream, some other entity is normally the sink for the stream. In this case, the normal possible external sinks include files, screens (or windows on screens), devices, and again, other programs. The notions of source and sink correspond to the notion of direction of flow for natural streams. The characters in a computer stream flow ***from*** the source ***to*** the sink.

Streams always have a beginning, but have an end only if they are finite. In prinicple, there is a sense of location, called the ***stream position***, for all streams. This sense of location, or stream position, corresponds to the natural notion of the point on the bank of the stream where one stands and watches the stream flow by. In the natural world, one can sometimes change the stream position by running along the bank, thereby either viewing an earlier portion of the stream (the water) or advancing to a later portion. Some kinds of character streams allow this sort of repositioning, while others do not.

A fundamental principle underlying the notion of stream is this:

> A program manipulating a stream need have no knowledge of what lies at the other end of the stream.

Thus, a program which is the source of a stream (is producing a stream) need have no knowledge of what is consuming the stream, and a program which is the sink for a stream (is consuming a stream) need have no knowledge of what is producing the stream.

Internally, a stream consists of an open source or sink of characters (e.g., a file, a socket, a window, etc.), a pointer to the next place to read or write, and a buffer for holding information until it's reasonable to transmit. The internal components illustrated in Figure 11 are the following:

- the (open) file;
- the pointer to the next place to read or write is called the *file pointer*. It is set to the beginning of the file when the file is opened. Whenever a read or write operation is performed on the file, the file pointer is moved.
- The buffer is used for efficiency. Reading or writing one character at a time from or to the disk (or most other sources or destinations of characters) would be very slow. Instead, for output, characters are written to the memory-resident buffer. The buffer is flushed when it reaches its capacity. Flushing causes the contents of the buffer to be written to disk. For input, blocks of characters are moved from the disk to the memory resident buffer. Prolog can then read the characters one at a time from the buffer much more efficiently

.

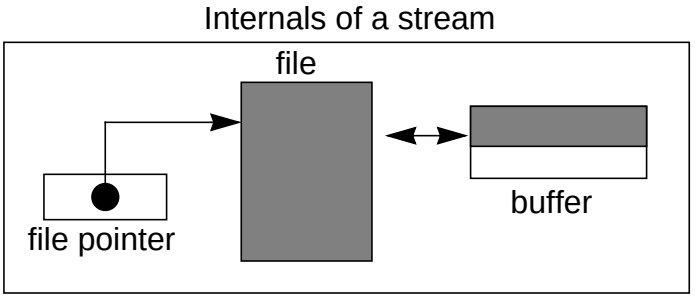


Figure 11. A Look Inside a Stream to a File.

The operations for dealing with streams fall into three groups:

- Operations for ***opening*** (or creating) streams.
- Operations for manipulating streams which are open (i.e., which exist).
- Operations for ***closing*** (or destroying) streams.

The only time a program must concern itself with what is at the other end of the stream is when it must open or create the stream. At this time, the program must specify what is to be at the other end of the stream. That is, it must specify what is to be the source or the sink for the stream. Thereafter, whether manipulating or closing the stream, the program need not worry about the nature or identity of the source or sink at the other end of the stream. It is the job of the underlying implementation of the stream facilities (provided by ALS Prolog) to take care of all of the details of moving the character stream between the program and the source or sink.

Under normal conditions, every ALS Prolog program automatically has two I/O character streams automatically opened for it by the underlying ALS Prolog system. These are called the ***default I/O*** streams. One of them is the default input stream and is normally connected to the keyboard, while the other is the default output stream, and is normally connected to the computer screen, or to a particular window on the screen when running under a graphical operating system. These will be discussed in more detail in a later section. All other streams which the program seeks to manipulate must be explicitly opened by the program, and should also be closed by the program when they are no longer required. Under normal circumstances, all

streams which remain open when an ALS Prolog program exits to the operating system are automatically closed.

All streams have both a ***mode*** and a ***type***. The ***mode*** of a stream basically reflects the direction of flow of information in the stream. The two fundamental stream modes are `read` and `write`. In `read` mode, the program is the consuming the stream's information, so that the program is the sink for the stream. In `write` mode, the program is producing the information on the stream, so that the program is the stream's source. Other modes are possible and will be discussed later.

The ***type*** of a stream reflects deeper information about the computing environment. The great majority of computing systems and programming languages (including earlier versions of Prolog and ALS Prolog) identify characters with bytes. However, this does not provide good support for alphabets with larger numbers of characters, and so a distinction is made between characters and bytes in the Prolog standard. Characters are internally represented as Prolog atoms, while bytes, as normally done, are represented by small integers. At bottom, a computer really moves bytes around, not characters. So at bottom, what a program thinks of as a stream of characters is really a stream of bytes. Since there is now a distinction between the notions of characters and bytes, a program can choose to view a data stream as either a stream of characters or as a stream of bytes. The type of a stream indicates this distinction. A ***text*** stream is a stream that is being viewed as a stream of characters. A ***binary*** stream is a stream that is being viewed as a stream of bytes. Facilities are provided for opening streams either as text or binary, and for manipulating them in both modes.

15.2 Opening and Closing Streams.

To open (or create) a stream, a program must indicate a ***source/sink*** for the stream (the other end from the program), must indicate a ***mode*** for the stream, and possible should indicate some ***options*** for the creation of the stream. Moreover, the process of opening or creating the stream should provide the program with some ***descriptor*** or handle with which to refer to the created stream for future manipulation.

15.2.1 Stream opening predicates.

There are two predicates used for opening streams:

open/3
open(SourceSink, Mode, Descriptor)
open(+, +, -)

open/4
open(SourceSink,Mode,Options,Descriptor)
open(+, +, +, -)

The three-argument version is simply definable in terms the four-argument version by:

```
open(SourceSink, Mode, Descriptor)
  :-
  open(SourceSink, Mode, [], Descriptor).
```

The arguments for `open/4` are described below.

15.2.2 SourceSink Terms.

A ***sourcesink*** term describes a target 'other end' of a stream. (The Prolog program of course holds on to 'this end' of the stream.) The particular sourcesink terms correspond to the various kinds of entities which can act as sources or sinks.

Files.

If the target source or sink is to be a file, the sourcesink term is a Prolog atom which is a name for the file, possibly including path information. (And conversely, an atom in the sourcesink position can only indicate a file as target source or sink.) As with `consult/1`, the file name may be either an abolute path name or a relative path name. (On most operating systems, the 'raw' keyboard and screen are handled more or less as special files. They will be discussed later.)

Strings.

Prolog strings (lists of characters) are acceptable sources and sinks for streams. When used as a source, a Prolog string `S` must be ground (fully instantiated), while when used as a sink, the string `S` must be an uninstantiated variable (which could be the tail of a larger list). In these cases, the sourcesink term is of the form

```
string(S)
```

When used as a sink, a string S (initially an uninstantiated variable) is viewed as a potentially infinite stream, which grows as characters are written to it.

Atoms and UIAs.

Atoms and UIAs are also acceptable sources and sinks for streams. In this case, the sourcesink term is of the form

```
atom(A)
```

where A is the atom or UIA in question. When used as a sink, the atom or UIA A is a stream of finite length, and any characters initially contained in A are gradually overwritten by the output process.

Sockets.

[Missing still: Tools for asyncronous, interrupt-driven sockets (e.g., for servers)].]

Sockets may also be used as source or sinks. The sourcesink term is of one of the following two forms:

```
socket(Target)   socket(DescriptionList)
```

In the first case, `Target` is any atom.. This first case is a shorthand for a particular instance of the second case, namely,

```
socket([target=Target]).
```

In the second case, which is the general case, `DescriptionList` is a list of ***equations***, each of which is of the form

```
Tag = Value.
```

The possible values `Tag` may take on are:

```
domain   type   protocol   port   target.
```

The expressions which `Value` may take on are determined by the value of `Tag`. The only required tag which must appear is the `target` tag. Default values are supplied for all other tags if no equation is present. The socket tags, the range of corresponding values, and their defaults, are described below.

domain: values = [af_unix, af_inet];

default = af_unix

type: values = [sock_stream, sock_dgram];

default = sock_stream

protocol: values = [0, ip, icmp, tcp, udp];

default = ip

port: values = ***any acceptable port number;***

default: sock_stream = 1599; sock_dgram = 1598.

target: values = ***an atom which is an acceptable machine name;***

When the socket is a read (incoming) socket, the null atom '' can be used.

Here are several simple examples:

```
socket(jarrett)
socket('')
socket([target='', domain=af_inet,
  type=sock_stream])
socket([target=jarrett, domain=af_inet,
  type=sock_dgram])
```

Windows.

When running windowed versions of ALS Prolog (e.g., the Motif version), text windows are acceptable sources and sinks for streams. In this case, the sourcesink term is of the form

```
window(Name)
```

where `Name` is an atom which is a name for the target window (see the Sections on windowing).

System V IPC queues.

On Unix systems configured to support System V IPC queues, these queues can be used as both sources and sinks for streams. In this case, the sourcesink term is of the form

```
sysV_queue(Key)
```

where `Key` is either an integer or an expression of the form `key(File, ID)`. Here, `File` and `ID` are appropriate inputs to the Unix `ftok` function which is used to obtain the IPC queue identifier. Specifically, `File` must be an atom which is a path to an ***existing*** file (possibly in */tmp*, etc) for which ***both*** processes which will communicate have appropriate access, typically both read and write. The second argument, `ID` is an arbitrary atom. Only the first character of this atom is significant: it is extracted by ftok and used in creating the IPC key. Both of the expressions below are examples of sourcesink descriptions for System V IPC queues:

```
sysV_queue(99)
sysV_queue(key('/tmp/foobar',mine))
```

15.2.3 Immediate versu Delayed Streams.

A file sourcesink is said to be ***immediate*** in the sense that (normally) all of the characters (data) which will make up the stream are immediately available once the stream is opened to the file. In contrast, a socket sourcesink is called ***delayed*** in the sense that access to some (possibly) all of the characters making up the stream may be delayed to the consuming process. We say that a stream is either immediate or dealyed if its corresponding sourcesink is immediate or delayed, respectively. This distinction is of particular significance for term-level I/O reading from this stream, but also has effects for character-level consumption of a delayed stream. Consider

the situation sketched in Figure 12 .

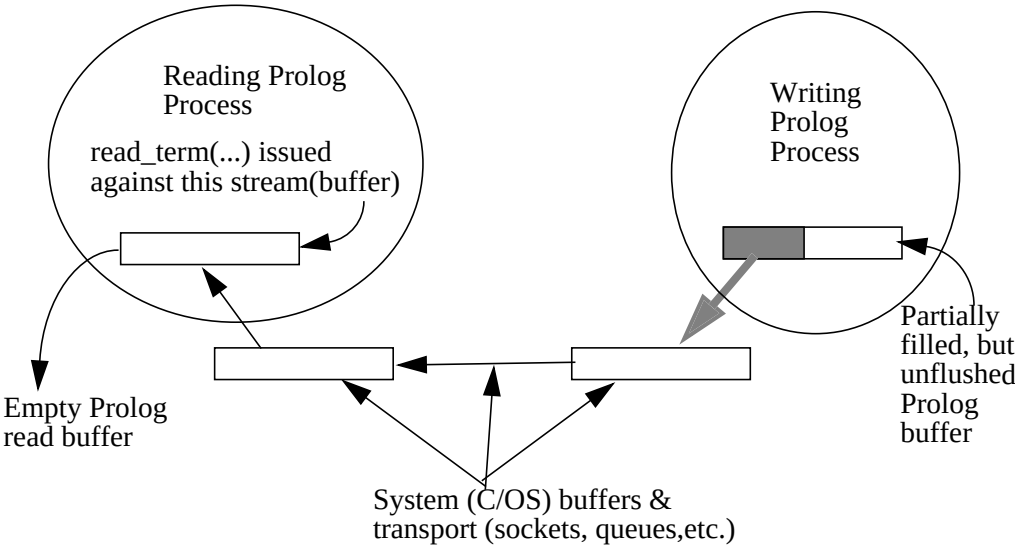


Figure 12. A Delayed Stream in a Delayed State.

When the Prolog system attempts to fill the read buffer (on the left) in response to the read_term call, it is unable to obtain any characters from the underlying OS machinery. If the sourcesink for the reading processes were a file, this situation would be interpreted as end of file. However, that is not what we want to do for a socket (for example), since the writing process will presumably fill and/or flush the buffer sooner or later, and more characters will become available to the reading process. The refinements we introduce for this situation are detailed below.

Character Input from Delayed Streams.

`get_char/2` either returns (the code of) a character, which is a non-negative integer, or returns -1 to signify end of file. We extend `get_char/2` to return -2 when the stream is a delayed stream which is still open, the stream has not reached end of stream, but no characters are available for processing.

Term Input from Delayed Streams.

The behavior of `read_term/3` for a delayed stream is governed by the read op-

tion `blocking(Bool)`, where `Bool` is either `true` or `false`. Note that the default behavior is `blocking(true)` -- a blocking read. Under the default `blocking(true)` option, `read_term/3` behaves just as it does for immediate streams. The difference in behavior occurs for delayed streams. Suppose that the situation in Figure 12 came about as follows. The `read_term(...)` goal was initiated, and at that time, some charazcters were available. But before `read_term` can finish parsing a complete Prolog term, the characters are consumed. If the stream were an immediate stream, we only run out of characters at end of stream, and in that case, the system would raise an exception, indicating that `end_of_file` (if it were a file) was encountered improperly.

However, when the stream S is a delayed stream which is in the delayed state of Figure 12 , the goal

```
read_term(S, T, [blocking(false)]),
```

succeeds, and binds `T` to the distinguished term

```
unfinished read.
```

The tokens which were read out of the stream's buffer in attempting this read_term are saved in the stream data structure. Subsequent calls

```
read_term(S, T, [blocking(false)]),
```

will attempt to again read from the stream, beginning with the tokens which were saved, and then continuing with any (possibly) new characters which have arrived since the last unfinished read.

15.2.4 Modes.

There are four possible modes for a stream:

```
read
write
read_write
append
```

However, only file streams support the read_write and append modes at the present time; all other streams only support the read or write modes.

The mode names rather clearly indicate the manner in which the streams to which

they apply will be used:

- A stream opened in `read` mode is being consumed by the program: the program is reading from the stream.
- A stream opened in `write` mode is being filled by the program: the program is writing to the stream. However, if the sink for this stream is a file which existed prior to the stream being opened, any previously existing contents of the file are discarded (i.e., the file is truncated).
- A stream opened in `read_write` mode is generally both read from and written to by the program; in general, this implies that the program's position in such a stream can be aribtrarily set and changed.
- Finally, `append` mode is like `write` mode, except that the previously existing contents of a file which is the sink for this stream are not lost: the stream position is automatically set to the end of the file and all output through the stream to the file is written at the end of the file.

15.2.5 Options.

The options argument of `open/4` is a list whose elements are acceptable stream open options. These provide additional information to the `open/4` procedure for refined control of the stream being created. These latter are differnt sorts of Prolog terms. Some of the options are applicable to all streams, while others apply only to streams connected to particular kinds or sources or sinks, or of particular modes or types.

text vs binary

At most one of `text` or `binary` can be present on the options list. When `text` is present, the stream is treated as a character stream. When `binary` is present, the stream is treated as a byte stream. If neither is present, the stream is treated as a character stream, so that the default is `text`.

aliases

An alias is an atom which acts as a global name for the stream to be opened. Once a stream has been opened, the stream descriptor (returned in the third argument of

open/3 and the fourth argument of open/4) can be passed around between procedures and used during stream manipulations (typically reads and writes). However, if an alias is assigned to the stream, the stream descriptor which is returned by open/[3,4] can be ignored, and all stream manipulations (e.g. reads and writes) can refer to the stream by using the alias instead of the stream descriptor. To indicate that Atom is to be an alias for a stream to be opened, the expression

```
alias(Atom)
```

should be included on the options list of the call to open/4 which creates the stream. (There are also procedures for dynamically assigning an alias to a stream after stream creation; these will be discussed later.)

seek_type

Some streams can be repositioned during program execution. These options indicate the type of repositionion which is requested for the stream. Not all streams support such repositioning. It the stream to be opened does not support the requested repositioning, the repositioning request generates an exception from open/4.. There are two types of repositioning requests:

```
seek_type(previous)
seek_type(byte)
```

The first indicates that the stream should support repositioning to any previously occupied position. The second indicates that the streams should support (re)positioning to any meaningful position.

[Note: The Prolog standard may cause the positioning options to change to:

```
reposition(true)
reposition(false)
```

The interpretations in the standard are:

```
reposition(true) -- same as seek_type(previous)
reposition(false) -- repositioning not requested
```

In this case, ALS Prolog will support the third value:

```
reposition(byte) -- same as seek_type(byte)        ]
```

buffering

These options allow the programmer to control the coordination between the stream's buffer and the ultimate external source or destination of the data. These options are most meaningful for file streams, but also have some significance for some other kinds of streams. A buffering option is indicated by a term of the form

```
buffering(BOption)
```

where `BOption` is one of the following three atoms:

```
byte
line
block
```

These options determine how often data is moved between the stream buffer and the source or destination. The first, `byte`, indicates that data should be moved on every character or byte (according as the stream is `text` or `binary`) handled by the program; in essence, this specifies no buffering should be used. The second, `line`, indicates that data should be moved on a line-by-line basis. The third option, `block`, essentially indicates that data should be moved in the largest units possible, as determined by the buffer size of the stream, and the block or buffer size of the external source or sink. The default is `block` buffering since it is generally the most efficient.

bufsize

This option allows the programmer to control the size of the buffer assigned to the stream. The option expressed by a term of the form

bufsize(N)

where N is an integer (which should be a power of 2). The default is 1024. The maximum value is determined by the largest contiguous area available on the Prolog heap.

prompt_goal

This option is useful when a pair of streams, one in `read` and one in write mode, are being used for interactive communication, say to a window, or to the keyboard and screen. However, this option strictly applies only to a single stream, and is

meaningful only if the stream is in `read` mode. The option is of the form

```
prompt_goal(Goal)
```

where Goal is a Prolog term which indicates a goal which can be run. Whenever the buffer of the stream to which this option applies (which must be in read mode) is empty, before attempting to refill the stream's buffer from the external source, the system will first execute the goal `Goal`. Here is how this option is normally used. Suppose we wish to use a read mode and write mode stream to window myWin, and that whenever the read stream buffer is empty, we want a prompt to be printed on the window. Consider the following code fragment:

```
open(window(myWin),write, [buffering(line),text],
  OutStream),
open(window(myWin), read,
       [prompt_goal(user_prompt_goal(OutStream))],
  InStream),
```

Both streams are opened with `line` buffering; the `write` stream is explicitly indicated to be a `text` stream, while the `read` stream is a `text` stream by default (this is just by way of example). In addition, the read stream has a prompt_goal option applied. Whenever the buffer of the read stream is empty, the system will first run the goal

get_user_prompt(Prompt),

put_atom(Stream,Prompt),

flush_output(Stream).

s are opened with `line` buffering; the `write` stream is explicitly indicated to be a `text` stream, while the `read` stream is a `text` stream by default (this is just by way of example). In addition, the read stream has a prompt_goal option applied. Whenever the buffer of the read stream is empty, the system will first run the goal

```
user_prompt_goal(OutStream).
```

The code defining this goal could be:

```
user_prompt_goal(Stream)
  :-
  put_atom(Stream, '>>'),
```

```
flush_output(Stream).
```

Here the desired prompt is `'>>'` . The I/O routines `put_atom/2` and `flush_output/1` will be described in later sections.

eof_action

[Not yet implemented.] For a stream which is opened in `read` or `read_write` mode, this option indicates what action the system should take if the program attempts to read beyond the end of the stream. The option is expressed by a term of the form

```
eof_action(Action)
```

where `Action` is one of the following atoms:

```
error
eof_code
reset
```

The interpretations of these action options are as follows:

`error:` An I/O end-of-file error exception is raised, signifying that no more input exists in this stream.

`eof_code:` The normal end-of-stream marker is returned (i.e., `end_of_file` for character and term input and -1 for character input).

`reset:` The stream is reset and another attempt is made to read from it.

write_eoln_type

This option allow the programmer to control which end-of-line (eoln) characters are output by nl/1. A write end-of-line option is indicated by a term of the form

```
write_eoln_type(Type)
```

where `Type` is one of the following three atoms:

```
cr
lf
crlf
```

These options determine what characters are output by `nl/1.` The first, `cr`, indicates that a carriage return (`"\r"`) should be output. The second, `lf,` indicates that a line feed (`"\n"`) should be output. The third, `crlf,` indicates that a carriage return followed by a line feed (`"\r\n"`) should be output.

The default for this option varies depending on the operating system being used. MacOS uses `cr`, unix systems use `lf`, and MS DOS and Win32 use `crlf`.

read_eoln_type

This option allows the programmer to control which characters should be used to detect an end-of-line (`eoln`) by `read/2` and `get_line/3.` A read end-of-line option is indicated by a term of the form

```
read_eoln_type(Type)
```

where Type is one of the following four atoms:

```
cr
lf
crlf
universal
```

These options determine what `read/2` and get_line/3 recognize as an end-of-line. The first, `cr`, indicates that a carriage return (`"\r"`) should be interpreted as an end-of-line. The second, `lf,` indicates that a line feed (`"\n"`) should be interpreted as an end-of-line. The third, `crlf,` indicates that a carriage return followed by a line feed (`"\r\n"`) should be interpreted as an end-of-line. The fourth, `universal`, indicates that any of the end-of-line types (`cr, lf, crlf`) should be interpreted as an end-of-line. The default is universal since this allows the correct end-of-line interpretation for text files on all operating systems.

Socket options

name = Name

connects(N)

System V IPC queue options

msg_type(T),

create

perms(_), perms(_,_,_)

15.2.6 Stream descriptors.

A ***stream descriptor*** is the term which is returned when a stream is opened (in the third argument of `open/3` and the fourth argument of `open/4`). It is a complex Prolog term. However, programmers should not attempt to 'look inside' of it nor attempt to rely on any of its structure. Appropriate predicates are supplied (see the following sections) for accessing and updating it as necessary. A stream descriptor is what the Prolog standard describes as ***implementation dependent***. As such, implementations are free to change the structure and nature of such terms. Conceivably, this could happen to stream descriptors in future releases of ALS Prolog. The stream descriptor is used simply as a means of referring to the stream in the predicates described in the following sections.

15.2.7 Closing Streams.

close/1.
close(Stream_or_Alias).
close(+).

Streams are very easily closed. One simply uses the unary predicate close/1 applied either to the stream descriptor or to an alias for the stream:

```
close(Stream_or_Alias).
```

15.3 Stream Environment.

The stream environment of an ALS Prolog program is made up of the collection of streams which are open, together with the special roles which have been assigned to some of them. This section describes this environment together with predicates for querying and altering its state (besides the basic predicates for opening and closing streams described in the last section), together with predicates for querying the state(s) of individual streams.

15.3.1 Standard Streams.

Two streams are automatically opened for every ALS Prolog program. Both are `text` streams, and one is in `read` mode while the other is in `write` mode. The `read` mode stream is automatically assigned the alias `user_input` while the write mode stream is automatically assigned the alias `user_output`. In addition, for compatibility with older versions of Prolog, both streams are (ambiguously) assigned the alias `user`.

When ALS Prolog is run as a TTY-style program under operating systems such as Unix or DOS, the `read` mode stream `user_input` is connected to the process's standard input (`stdin`), while the `write` mode stream `user_output` is connected to the process's standard output (`stdout`). When ALS Prolog is run in one of its windowing versions (e.g., Motif), both `user_input` and `user_output` are connected to the ALS Prolog Worksheet window.

15.3.2 Current Streams.

No matter what streams have been opened (whether automatically or by the program), ALS Prolog always maintains a notion of the current input and output streams. The current input and output streams serve as defaults for all of the input/ouput operations to be described in the following sections. Thus, if a a version of an I/O operation is used without a stream argument (e.g. a read or a write without a stream argument), the operation is applied to the appropriate current input or output stream.

When ALS Prolog starts up, the current input and output streams are automatically set to the standard input and output streams. However, the program can change the settings of the current input and output streams. The following predicates are used for manipulating the current streams.

current_input/1
current_input(Stream)
current_input(?)

The goal

```
current_input(Stream)
```

is true iff the stream `Stream` is the current input stream. Operationally, this means that `current_input` unifies `Stream` with the stream descriptor of the current input stream. Note that this means that `current_input` applies to stream descriptors. In particular, even if `Alias` is an alias for the current input stream, calling `current_input(Alias)` will fail.

current_output/1
current_output(Stream)
current_output(?)

The goal

```
current_output(Stream)
```

is true iff the stream `Stream` is the current output stream. Operationally, this means that `current_output` unifies `Stream` with the stream descriptor of the current output stream. Note that this means that `current_output` applies to stream descriptors. In particular, even if `Alias` is an alias for the current output stream, calling `current_output(Alias)` will fail.

set_input/1
set_input(Stream_or_alias)
set_input(+)

`set_input/1` is used to set the current input stream. If `Stream_or_alias` is instantiated to either a stream descriptor of a stream in either read or read_write mode, or is instantiated to an alias for such a stream, then a call to `set_input(Stream_or_alias)` changes the current input stream to be the stream associated with `Stream_or_alias`. If `Stream_or_alias` is inappropriate for any reason, `set_input/1` raises an exception. Thus `set_input(S_or_a)` cannot fail. Either it succeeds or it raises an exception, in which case the current input stream remains unchanged.

set_output/1
set_output(Stream_or_alias)
set_output(+)

`set_output/1` is used to set the current output stream. If `Stream_or_alias` is instantiated to either a stream descriptor of a stream in ei-

ther write, read_write, or append mode, or is instantiated to an alias for such a stream, then a call to

```
set_output(Stream_or_alias)
```

changes the current output stream to be the stream associated with `Stream_or_alias`. If `Stream_or_alias` is inappropriate for any reason, `set_output/1` raises an exception. Thus `set_output(S_or_a)` cannot fail. Either it succeeds or it raises an exception, in which case the current output stream remains unchanged. .

15.3.3 Stream Charcteristics

stream_property/2
stream_property(Stream, Property)
stream_property(?, ?)

`stream_property/2` is used to determine whether or not a given property applies to a given stream; It can also be used to enumerate or generate the (open) streams possessing a certain property. Declaratively,

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

is true if and only if `Property` holds of `Stream`. The possible values for `Property` include all of the expressions which may appear on the `Options` list passesd to `open/4`, together with the following:

```
input, output, and  mode(M),
```

where `M` is one of `text, binary`.

From a more procedural point of view, the action of `stream_property` is as follows. If both `Stream` is instantiated to the stream descriptor of a currently open stream, and `Property` is instantiated to a property descriptor which is true of `Stream`, then

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

succeeds. Either or both of `Stream` or `Property` may be uninstantiated. In this case,

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

is resatisfiable under backtracking. Thus, the action of in this case is effectively to compute the pairs S,P such that S is a currently open stream which has property P, in some undetermined order, and to unify `Stream` with S and `Property` with P.

For example, consider the goal

```
stream_property(S, output).
```

If S is instantiated to a stream descriptor, this goal will check whether output is permitted on this stream. If S is uninstantiated, under backtracking, S will successively be instantiated to all streams currently open for output.

As another example, consider the goal

```
stream_property(S, file_name(F)).
```

If S is instantiated to a stream descriptor, then if the source or sink for S is a file, F will be unified with the name of the file to which S is connected; otherwise, the goal will fail. If S is uninstantiated, but F is instantiated to an atom (which is the name of a file), then if some currently open stream has file F as its source or sink, S will be unified with that stream's descriptor. Finally, if both S and F are uninstantiated, this goal, under backtracking, will compute all the pairs S,F where F is a file name and S is a currently open stream connected to F.

When used in non-determinate ways, stream_property exhibits a "logical" semantics for state changes of the stream environment. For example, consider the goal

```
stream_property(S,P), write(S:P), nl, close(S), fail.
```

This goal will enumerate all the properties for all streams which were open before this goal was run. Note that this example may call `close(S)` several times for each stream S, but this does not cause any problem since `close` simply succeeds if called on a stream which is already closed.

is_stream/2
is_stream(Stream_or_alias, Stream)
is_stream(+, ?)

Succeeds when `Stream_or_alias` is a stream or alias and will bind `Stream` to the corresponding stream.

assign_alias/2.
assign_alias(Alias, Stream_or_alias)
assign_alias(+, +)

If `Stream_or_alias` is associated with the stream S, and if `Alias` is a term, associates `Alias` to S as an alias. Note that a given stream can carry more than one alias, and that `Alias` can be a compound term.

cancel_alias/1.
cancel_alias(Alias)
cancel_alias(+)

If `Alias` is currently associated with stream S as an alias, removes the alias association between `Alias` and S.

reset_alias/2.
reset_alias(Alias, Stream_or_alias)
reset_alias(+, +)

If `Alias` is currently associated with stream S as an alias, and if `Stream_or_alias` is associated with stream S', first removes the association between `Alias` and stream S, and then associates `Alias` to stream S' as an alias.

current_alias/2.
current_alias(Alias,Stream)
current_alias(?,?)

Succeeds iff `Alias` is an alias which is associated with the stream `Stream`.

15.3.4 Stream Positions

at_end_of_stream/0
at_end_of_stream/1
at_end_of_stream(Stream_or_alias)
at_end_of_stream(+)

Consider a stream `S` which has been opened for input. If the stream `S` is of finite length, it is possible to reach a state in which all the characters or bytes in the stream `S` have been read by input routines

(such as `get_byte, get_code, get_char, or read`),

or when `set_stream_position/2` has been used to move directly to the end of the stream. At such a point, it is still valid to call an input routine. Each of the input routines returns a specific value to indicate that end of stream has been reached: `get_code` and `get_byte` return -1; `get_char` and `read` each return the atom `end_of_file`. When one of these terminating values has been read, the stream is said to be ***past*** the end of the stream, while the stream is said to be ***at*** the end of stream when no more characters or bytes are available for input, but no such input call has been made.

```
at_end_of_stream(Stream_or_alias)
```

succeeds if and only the stream associated with `Stream_or_alias` is either at end of stream or is past end of stream.

The predicate `at_end_of_stream/0` determines whether the current_input stream is at end of stream The predicate `at_end_of_stream(Stream)` still succeeds when called in the past end of stream state. A stream need not have an end, in which case this predicate would never succeed for that stream. If the source for `S_or_a` is a device such as a terminal, and if there is no input currently available on that device, then `at_end_of_stream` will wait for input just as get_code would do.

at_end_of_line/0.
at_end_of_line/1.
at_end_of_line(Alias_or_stream)
at_end_of_line(+)

These predicates determine whether a stream is at positioned at the end of a line, in an elementary way. They simply attempt to perform a `peek_char` on the stream, and determine if the character returned is identical with newline (i.e., the character with code 0'\n). The are defined by:

```
at_end_of_line :-
    get_current_input_stream(Stream),
    at_end_of_line(Stream).

at_end_of_line(Alias_or_stream) :-
```

```
peek_char(Alias_or_stream,0'\n).
```

flush_output/0
flush_output/1
flush_output(Stream_or_alias)
flush_output(+)

If the stream associated with `Stream_or_alias` is a currently open output stream, then any output which is currently buffered by the system for that stream is (physically) sent to that stream, and `flush_output(Stream_or_alias)` succeeds.

`flush_output/0` flushes the current output stream; it is defined by

```
flush_output :-
  current_output(Stream),
  flush_output(Stream.
```

flush_input/0
flush_input/1
lush_input(Stream_or_alias)
flush_input(+)

If the stream associated with `Stream_or_alias` is a currently open input stream, then any input which is currently buffered by the system for this stream is discarded, and `flush_input(Stream_or_alias)` succeeds.

`flush_input/0` flushes the current input stream; it is defined by

```
flush_input :-
  current_input(Stream),
  flush_input(Stream..
```

stream_position/3
stream_position(Stream_or_alias, Current_position, New_position)
stream_position(+, ?, ?)

If the stream associated with `Stream_or_alias` supports repositioning, then the call to `stream_position/3` causes `Current_position` to be unified with the current stream position of the stream, and, as a side effect, the stream

position of this stream is set to the position represented by `New_position`. `New_position` may be one of the following values:

- An absolute integer which represents the address of a character in the stream. The beginning of the stream or the first character in the stream has position 0. The second character in the stream has position 1 and so on.
- The atom `beginning_of_stream`.
- The term `beginning_of_stream(N)` where `N` is an integer greater than zero. The position represented by this term is the beginning of the stream plus `N` bytes.
- The atom `end_of_stream`.
- The term `end_of_stream(N)` where `N` is an integer less than or equal to zero. This allows positions (earlier than the end of stream) to be specified relative to the end of the stream. The position represented by this term is the end-of-stream position plus `N` bytes.
- The atom `current_position`.
- The term `current_position(N)` where `N` is an integer. This allows positions to be specified relative to the current position in the file.

set_stream_position/2
set_stream_position(Stream_or_alias, Position)
set_stream_position(+, +)

`set_stream_position(Stream_or_alias, Position)` changes the position of the stream associated with `Stream_or_alias` to `Position`. This predicate is effectively defined by:

```
set_stream_position(Stream_or_alias, Position) :-
  stream_position(Stream_or_alias, _, Position).
```

The possible error exceptions are the same as those for `stream_position/3`.

15.4 Byte Input/Output.

get_byte/1

get_byte(Byte)
get_byte(?)

Unifies Byte with the next byte obtained from the current input stream.

get_byte/2
get_byte(Alias_or_stream, Byte)
get_byte(+, ?)

Unifies `Byte` with the next byte obtained from the stream associated with `Alias_or_stream`.

put_byte/1
put_byte(Byte)
put_byte(Byte)

Outputs the byte `Byte` to the current output stream.

put_byte/2
put_byte(Alias_or_stream,Byte)
put_byte(Alias_or_stream,Byte)

Outputs the byte `Byte` to the stream associated with `Alias_or_stream`.

15.5 Character Input/Output.

The predicates in this section perform character-level input and output on streams. While these predicates are primarily inteded for use on streams opened in text mode, they have meaning for streams opened in binary mode, unless otherwise indicated. There are related byte-oriented predicates for streams opened in binary mode. (Note that ALS Prolog is more relaxed than the ISO standard; the latter states that character operations cannot be performed on binary streams, and that byte operations cannot be performed on character streams)

get_char/1
get_char(Char)
get_char(?)

get_char/2

get_char(S_or_a, Char)
get_char(+, ?)

`get_char(Char)` is equivalent to `get_char(S, Char)` where `S` is the current input stream; that is, `get_char/1` is effectively defined by:

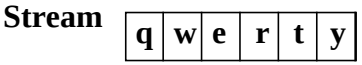
```
get_char(Char) :-
  current_input(Stream)
  get_char(Stream, Char).
```

Let `Stream_or_alias` be properly instantiated, and let S be the stream associated with `Stream_or_alias`. Then if S is a text stream,

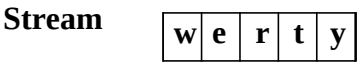
```
get_char(Stream_or_alias, Char)
```

is true iff `Char` unifies with the next character to be read from S, while if S is a binary stream, it is true iff `Char` unifies with the next byte to be read from S. If S is at or past end of stream, the 'eof action' associated with S is performed. If S is a delayed stream, and is in the delayed state when the goal is issued, `Char` is bound to -2.

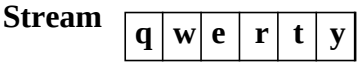
Examples



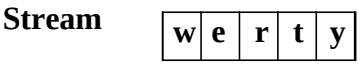
get_char(Stream, CC)



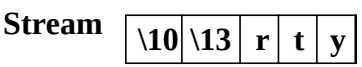
Success: **CC ⟶ q**



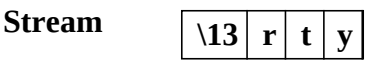
get_char(Stream, p)



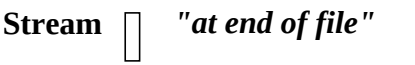
Fail



get_char(Stream, CC)



Success: **CC ⟶ \10**



get_char(Stream, CC)

Stream ***"past end of file"***

Success: **CC ⟶ end_of_file**

get_nonblank_char/1

get_nonblank_char(Char)
get_nonblank_char(-)

get_nonblank_char/2
get_nonblank_char(Stream, Char)
get_nonblank_char(+, -)

`get_nonblank_char/1` is defined by

```
get_nonblank_char(Char) :-
    get_current_input_stream(Stream),
    get_nonblank_char(Stream,Char).
```

`get_nonblank_char(Stream, Char)` unifies `Char` with the next non-whitespace character obtained from the input stream associated with `Alias_or_stream`, if such a character occurs before the next end of line, and unifies `Char` with the atom

```
end_of_line
```

otherwise.

get_atomic_nonblank_char/1
get_atomic_nonblank_char(Char)
get_atomic_nonblank_char(-)

get_atomic_nonblank_char/2
get_atomic_nonblank_char(Stream, Char)
get_atomic_nonblank_char(+, -)

`get_atomic_nonblank_char/1` is defined by

```
get_atomic_nonblank_char(Char) :-
    get_current_input_stream(Stream),
    get_atomic_nonblank_char(Stream,Char).
```

`get_atomic_nonblank_char(Stream, Char)` unifies `Char` with the atomic form of the next non-whitespace character obtained from the input stream associated with `Alias_or_stream`, if such a character occurs before the next end of line, and unifies `Char` with the atom

```
end_of_line
```

otherwise. It is defined by

```
get_atomic_nonblank_char(Stream,Char)
    :-
    get_nonblank_char(Stream,Char0),
    (Char0 = end_of_line ->
        Char0 = Char
        ;
        name(Char, [Char0])
    ).
```

peek_char/1.
peek_char(Char)
peek_char(-)

peek_char/2
peek_char(Alias_or_Stream, Char)
peek_char(+, -)

`peek_char(Char)` unifies `Char` with the next character obtained from the default input stream. However, the character is not consumed.

`peek_char(Alias_or_Stream, Char)` unifies `Char` with the next character obtained from the stream associated with `Alias_or_Stream`. However, the character is not consumed.

put_char/1
put_char(Char)
put_char(+)

put_char/2
put_char(Stream_or_alias, Char)
put_char(+,+)

`put_char(Char)` is equivalent to `put_char(S, Char)` where S is the current output stream; that is, `put_char/1` is effectively defined by

```
put_char(Char) :-
  current_output(Stream),
```

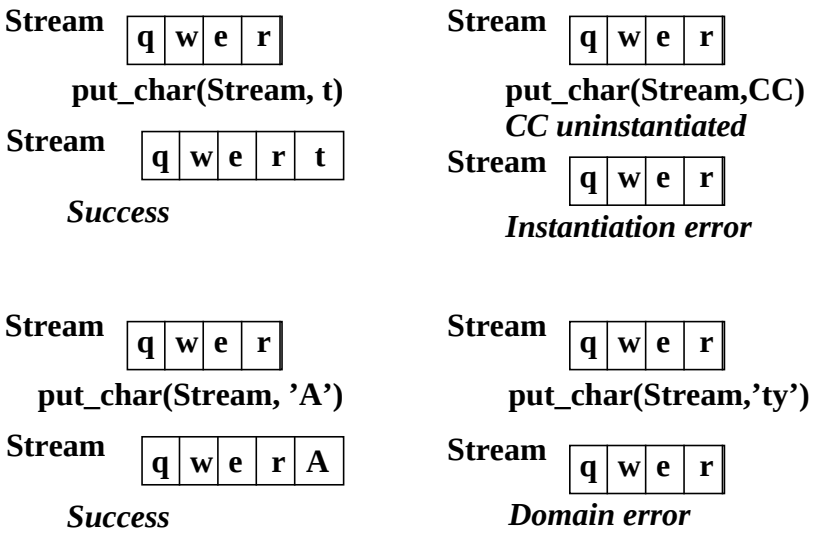
```
put_char(Stream, Char).
```

If `Stream_or_alias` is properly instantiated, S is the stream associated with `Stream_or_alias`, and `Char` is a character, then

```
put_char(Stream_or_alias, Char)
```

outputs the character `Char` to S, and changes the stream position on S to take account of the character which has been output.

Examples



put_string/1
put_string(String)
put_string(+)

put_string/2
put_string(Alias_or_stream, String)
put_string(+, +)

`put_string(String)` is defined by

```
put_string(String) :-
```

```
get_current_output_stream(Stream),
put_string(Stream, String).
```

If `String` is a Prolog string (i.e., a list of character codes), `put_string(Alias_or_stream, String)` recursively applies put_char to `String` to output the characters associated with `String` to `Alias_or_stream`.

put_atom/2
put_atom(Alias_or_stream,Atom)
put_atom(+,+)

Outputs the atom `Atom` to the stream associated with `Alias_or_stream`. Very efficient.

get_number/3
get_number(Alias_or_stream,InputType,Number)
get_number(+,+,?)

Attempts to read a number of the given type `InputType` from the stream associated with `Alias_or_stream`, and if successful, unifies the result with `Number`. The possible values for `InputType` are:

InputType	Type of Number
byte	signed byte (8 bit)
ubyte	unsigned byte (8 bit)
char	signed byte (8 bit) (synonymous with byte)
uchar	unsigned byte (8 bit) (synonymous with byte)
short	signed short integer (16 bit)
ushort	unsignged short integer (16 bit)
int	signed integer (32 bit)
uint	unsignged integer (32 bit)
long	signed integer (32 bit)
ulong	unsignged integer (32 bit)

InputType	Type of Number
float	floating point (32 bit)
double	floating point (64 bit)

put_number/3
put_number(Stream_or_alias,OutputType,Number)
put_number(+,+,+)

`put_number(Alias_or_stream,OutputType,Number)` outputs the number `Number` as `OutputType` to the stream associated with `Stream_or_alias`. `OutputType` may take on the following values:

```
byte   short   long   float   double.
```

get_line/1
get_line(Line)
get_line(?)

`get_line(Line)` is defined by

```
get_line(Line) :-
    get_current_input_stream(Stream),
    get_line(Stream,Line).
```

get_line/2
get_line(Stream_or_Alias, Line)
get_line(+, ?)

Reads the current line or remaining portion thereof from the stream associated with `Stream_or_Alias` into a UIA, and unifies this UIA with `Line`. If end-of-file is encountered before any characters, this predicate will fail. If end-of-file is encountered before the newline, then this predicate will unify `Line` with the UIA containing the characters encountered up until the end-of-file.

put_line/1
put_line(Line)
put_line(+)

put_line/2
put_line(Stream,Line)
put_line(+,+)

`put_line(Line)` is defined by

```
put_line(Line) :-
    get_current_output_stream(Stream),
    put_line(Stream,Line).
```

`put_line(Stream, Line)` is defined by

```
put_line(Stream,Line) :-
    put_atom(Stream,Line),
    nl(Stream).
```

skip_line/0
skip_line/1
skip_line(Alias_or_stream)
skip_line(+)

skip_line/0 is defined by

```
skip_line :-
    get_current_input_stream(Stream),
    skip_line(Stream).
```

If `Alias_or_stream` is open for (text) input, `skip_line(Alias_or_stream)` skips to the next line of input for the stream associated with `Alias_or_stream`.

nl/0
nl/1
nl(Stream_or_alias)
nl(+)

`nl` is equivalent to `nl(S)` where S is the current output stream; that is,

```
nl :-
  current_output(Stream),
  nl(Stream).
```

`nl(Stream_or_alias)` causes the current line or record on the stream associated with `Stream_or_alias` to be terminated.

15.6 Character Code Input/Output

These predicates provide a means of directly manipulating streams at the character code level.

get_code/1
get_code(Code)
get_code(?)

get_code/2
get_code(Stream_or_alias, Int)
get_code(+,?)

`get_code(Code)` is equivalent to `get_code(S, Code)` where `S` is the current input stream; i.e., `get_code/1` is defined by:

```
get_code(Code) :-
  current_input(Stream),
  get_code(Stream,Code).
```

Assume that `Stream_or_alias` is instantiated to a stream descriptor or to the alias of a stream descriptor, and let `S` be the stream associated with `Stream_or_alias`. If `S` is a text stream then

```
get_code(Stream_or_alias, Code)
```

is true iff `Code` unifies with the character code corresponding to the next character to be read from `Stream`, else if `S` is a binary stream it is true iff `Code` unifies with the next byte to be read from `S`.

put_code/1
put_code(Code)
put_code(+)

`put_code(Code)` is equivalent to `put_code(S, Code)` where `S` is the current output stream; i.e., `put_code/1` is defined by:

```
put_code(Code) :-
```

```
current_output(Stream),
put_code(Stream, Code).
```

put_code/2
put_code(Stream_or_alias, Code)
put_code(+,+)

`put_code(Stream_or_alias, Code)` outputs the character with code `Code` to the stream associated with `Stream_or_alias`, and changes the stream position on the stream associated with `Stream_or_alias` to take account of the character which has been output.

15.7 Term Input/Output.

ALS Prolog provides a rich array of predicates for I/O at the term level. The predicates in this section provide means for reading Prolog terms from input streams and for writing Prolog terms to output streams. For both input of terms and output of terms, there are a variety of options which can be requested. These options are controlled through the use of options lists which are passed to the various term-level I/O predicates.

A ***read options list*** is a list of read options, where the possible read options are defined as follows. Let T be the term which is (to be) read from the input stream.

variables(Vars)	Vars is unified with a list of the variables encountered in a left to right traversal of the term T.
variable_names(VNs)	VNs is unified with a list of elements of the form V=A, where V is variable which occurs in the term T and A is the associated name of that variable; the elements V=A occur in the order in which the variables V are encountered in the term T when T is scanned left to right.
vars_and_names(Vars,Names)	Vars is unified with a list of the variables encountered in a left to right traversal of the term T; Names is a list of the associated names of the vari-

ables. '_' is the only variable name which may occur more than once on the list Names.

singletons(Vars,Names)

Vars is unified with the singleton variables (those occuring only once) in the term T; Names is unified with a corresponding list consisting of the names of those singleton variables. Variables with name '_' are excluded from these lists.

syntax_errors(Val)

Val indicates how the system is to handle any syntax errors which occur durin g the reading of T. The possible values of Val and their interpretation are:

error Occurrence of a sysntax error will cause the system to raise an exception, which includes outputing a warning message. This is the default.

fail Occurrence of a syntax error will cause the attempt to read T to fail, and and error message will be output.

quiet Occurrence of a syntax error will cause the attempt to read T to fail quietly, with no message output.

dec10. Occurrence of a syntax error will cause the attempt to read T to output an error message, to skip over the offending input charaters, and attempt to re-read T from the source stream.

blocking(Bool)

For streams, such as socket streams or IPC queue streams, it is possible for the stream to remain open, yet there be no characters available at the time a read is issued. The value of Type controls the behavior of the read in this setting. The values of Type are as follows:

true In this type of read, the read suspends or waits until enough characters are available to parse as a valid Prolog term.

false ***In this type of read, if no characters are available, the read will immediately return with success, unifying the 'read term argument' (which is argument 1 for read_term/2 and is argumen 2 for read_term/3) with the term*** `unfinished_read`***; any tokens consumed in the read attempt are saved in the stream data structure, and subsequent attempts to read from this stream begin with these tokens, followed by tokens created from further characters which arrive at later times.***

debugging

This option is not handled in satisfy_options/4. Read terms are considered to be clauses and debugging information is attached. This is likely to of little direct use to the application programmer.

attach_fullstop(Bool)

This option determines if a fullstop is | added to

the tokens comprising a the term to be read. It is most | useful when used in conjunction with atom or list streams.

read_term/2
read_term(Term, Options)
read_term(?, +)

`read_term(Term, Options)` is equivalent to `read_term(S, Term, Options)` where S is the current input stream; i.e., `read_term/2` is defined by:

```
read_term(Term, Options) :-
  current_input(Stream),
  read_term(Stream, Term, Options).
```

read_term/3
read_term(Stream_or_alias, Term, Options)
read_term(+, ?,+)

`read_term(Stream_or_alias, Term, Options)` inputs a sequence TT of tokens from the stream associated with `Stream_or_alias` until an end token has been read. It is a syntax error if end of stream is reached before an end token is found. TT is then parsed as a Prolog term which is then unified with `Term`.

A 'sequence of tokens' implies that single quotes and double quotes (if included in the standard) are balanced. That is, an apparent end token appearing inside any kind of quotes is not an end token. However, parentheses and square brackets need not be balanced. The effect of this predicate may be modified by clauses of the special user-defined procedure `char_conversion/2.`

read/1
read(Term)
read(?)
read/2
read(Stream_or_alias, Term)
read(+,?)

`read(Term)` is equivalent to `read_term(S, Term, [])` where S is the current input stream; that is, `read/1` is defined by:

```
read(Term) :-
  current_input(Stream),
  read_term(Stream, Term, []).
```

`read(Stream_or_alias, Term)` is equivalent to

```
read_term(Stream_or_alias, Term, []),
```

so that `read/2` is defined by:

```
read(Stream_or_alias, Term) :-
  read_term(Stream_or_alias, Term, []).
```

The following convenience predicates are quite useful:

atomread/2
atomread(Atom,Term)
atomread(+,-)

atomread/3
atomread(Atom,Term,Options)
atomread(+,-,+)

bufread/2
bufread(String,Term)
bufread(+,-)

bufread/3
bufread(String,Term,Options)
bufread(+,-,+)

These are defined as follows:

```
atomread(Atom,Term)
    :-
    atomread(Atom,Term,[]).

atomread(Atom,Term,Options)
```

```
    :-
    open(atom(Atom),read,Stream),
    read_term(Stream,Term,
                [attach_fullstop(true)|Options]),
    close(Stream).
bufread(String,Term)
    :-
    bufread(String,Term,[]).

bufread(String,Term,Options)
    :-
    open(string(String),read,Stream),
    read_term(Stream,Term,
                [attach_fullstop(true)|Options]),
    close(Stream).
```

Just as the reading of terms from input streams can be affected by read options, the writing of terms to ouptut streams can be controlled by write options. The terms written are output in a pretty-printed format which breaks lines before wrapping, and uses indenting for legibility. Also, by default, the variables occurring in a term are represented using identifiers of the forms A, ..., Z, A1,....,A2,..... A ***write options list*** is a list of write options, which are one of the terms defined below. The default setting for each of these options is indicated in square brackets following each write option term. Let T be the term being written out, and let the expression Bool take on one of the values `true` or `false`.

quoted(Bool) [default: Bool = false]

If Bool = true, forces all symbols in T to be written out in such a manner that read_term/[2,3] may be used to read them back in. Bool = false indicates that symbols should be written out without any special quoting. In the latter case, embedded control characters will be written out to the output device as is.

ignore_ops(Bool) [default: Bool = false]

If Bool = true, operators in T will be output in function notation (i.e., operators are ignored.) If Bool = false, operators will be printed out appropriately.

portrayed(Bool) not implemented yet

numbervars(Bool) [default: Bool = true]

If Bool = true, terms of the form $VAR(N) where N is an integer will print out as a letter.

lettervars(Bool) [default: Bool = true]

If Bool = true, variables occurring in T will be printed out as letters A, B, ..., or letters followed by numbers: A1,...,A2,... If Bool = false, variables will be printed as _N where N is computed via the internal address of the variable. This latter mode will be more suited to debugging purposes where correspondences between variables in various calls is required. maxdepth(N,Atom1,Atom2) [default: maxdepth(20000,*,...)] N is the maximum depth to which to print; Atom1 is the atom to output when this maximum depth has been reached in printing any term. Atom2 is the atom to output when this depth has been reached at the tail of a list.

maxdepth(N) [default: N = 20000]

Equivalent to maxdepth(N,*,...).

line_length(N) [default: N = 78]

N is the length in characters of the output line. The pretty printer will attempt to put attempt to break lines before they exceed the given line length.

indent(N) [default: N = 0] N is the initial indentation to use.

quoted_strings(Bool) If Bool = true, lists of suitably small integers will print out as a double quoted string. If Bool = false, these lists will print out as lists of small numbers.

depth_computation(Val) [default: Val = nonflat]

Val may be either flat or nonflat. This setting determines the nature of the pretty printer's "depth in term" computation. If Val = flat, all arguments of a term or list will be treated as being at the same depth. If Val = nonflat, then each subsequent argument in a term (or each sebsequent element of a list) will be considered to be at a depth one greater than the depth of the preceding structure argument (or list element).

write_term/2
write_term(Term, Options)
write_term(+, +)
write_term/3
write_term(Stream_or_alias, Term, Options)
write_term(+,+,+)

`write_term(Term, Options)` is equivalent to `write_term(Out, Term, Options)` where `Out` is the current output stream; that is, `write_term/2` is defined by:

```
write_term(Term, Options) :-
```

```
    current_output(Stream),
    write_term(Stream, Term, Options).
```

`write_term(Stream_or_alias, Term, Options)` outputs `Term` to the stream associated with `Stream_or_alias` in a form which is defined by the write-options list `Options`.

Examples

write_term(S, [1,2,3]) → [1,2,3]

write_term(S, [1,2,3], [ignoreops(true)]) → . (1, . (2, . (3, [])))

write_term(S, '1 < 2') → 1 < 2

write_term(S, '1 < 2', [quoted(true)]) → '1 < 2'

write_term(S, 'VAR'(0),[numbervars(true)]) → A

write_term(S, 'VAR'(1),[numbervars(true)]) → B

write_term(S, 'VAR'(28),[numbervars(true)]) → C1

write/1
write(Term)
write(+)
write/2
write(Stream_or_alias, Term)
write(+,+)

`write(Term)` is equivalent to `write(Out, Term)` where `Out` is the current output stream; that is, `write/1` can be defined by:

```
write(Term) :-
  current_output(Stream),
  write(Stream, Term).
```

`write(Stream_or_alias, Term)` is equivalent to

```
write_term(Stream_or_alias, Term,
  [numbervars(true)]);
```

Thus, `write/2` can be defined by:

```
write(Stream_or_alias, Term) :-
  write_term(Stream_or_alias, Term,
  [numbervars(true)]).
```

Examples

write(S, [1,2,3]) ⟶ [1,2,3]
write(S, 1 < 2) ⟶ 1 < 2
write(S, 'VAR'(0) < 'VAR(1)) ⟶ A < B

writeq/1
writeq(Term)
writeq(+)
writeq/2
writeq(Stream_or_alias, Term)
writeq(+,+)

`writeq(Term)` is equivalent to `writeq(Out, Term)` where `Out` is the current output stream; i.e, `writeq/1` can be defined by:

```
writeq(Term) :-
  current_output(Stream),
  writeq(Stream, Term).
```

`writeq(Stream_or_alias, Term)` is equivalent to

```
write_term(Stream_or_alias, Term, [quoted(true),
  numbervars(true)]);
```

that is, `writeq/2` can be defined by:

```
writeq(Stream_or_alias, Term) :-
  write_term(Stream_or_alias, Term,
                  [quoted(true), numbervars(true)]).
```

Examples

writeq(S, [1, 2, 'A']) → [1, 2, 'A']

writeq(S, '1 < 2') → '1 < 2'

writeq(S, 'VAR'(0) < 'VAR(1)) → A < B

write_canonical/1

write_canonical(T)
write_canonical(+)
write_canonical/2
write_canonical(Stream_or_alias, Term)
write_canonical(+,+)

`write_canonical(T)` is equivalent to `write_canonical(S, T)` where S is the current output stream; that is, `write_canonical/1` can be defined by:

```
write_canonical(T) :-
  current_output(Stream),
  write_canonical(Stream, T).
```

`write_canonical(Stream_or_alias, Term)` is equivalent to

```
write_term(Stream_or_alias, Term, [quoted(true),
  ignore_ops(true)]);
```

that is, `write_canonical/2` can be defined by:

```
write_canonical(Stream_or_alias, Term) :-
  write_term(Stream_or_alias, Term,
                  [quoted(true), ignore_ops(true)]).
```

Examples

write_canonical(S, [1, 2, 3]) → '.'(1, '.'(2, '.'(3, [])))

write _canonical(S, 1 < 2) → < (1, 2)

write _canonical(S, '1 < 2') → ' 1 < 2 '

write_canonical(S, 'VAR'(0) < 'VAR'(1)) → < ('VAR' (0), 'VAR' (1))

write_clause/1.
write_clause(Clause)
write_clause(+)

write_clause/2.
write_clause(Alias_or_stream, Clause)
write_clause(+, +)

write_clause/3.
write_clause(Alias_or_stream, Clause, Options)
write_clause(+, +, +)

These convenience predicates are defined by:

```
write_clause(Clause) :-
    get_current_output_stream(Stream),
    write_clause(Stream, Clause).
```

And:

```
write_clause(Stream, Clause) :-
    write_clause(Stream, Clause, []).
```

And:

```
write_clause(Stream, Clause, Options) :-
    write_term(Stream,Clause, Options),
    put_char(Stream, 0'.),
    nl(Stream).
```

Since one often must ouput sequences of clauses, the following predicates are useful:

write_clauses/1.
write_clauses(Clauses)
write_clauses(+)

write_clauses/2.
write_clauses(Alias_or_stream, Clauses)
write_clauses(+, +)

These predicates are defined by:

```
write_clauses(Clauses) :-
    get_current_output_stream(Stream),
    write_clauses(Stream, Clauses, []).

write_clauses(Stream, Clauses) :-
    write_clauses(Stream, Clauses, []).
```

write_clauses/3
write_clauses(Alias_or_stream, Clauses, Options)
write_clauses(+, +, +)

If `Clauses` is a list of terms (to be viewed as clauses), `write_clauses/3` recursively applies `write_clause/3` to the elements of `Clauses`.

printf/1
printf(Format)
printf(+)

printf/2
printf(Format,ArgList)
printf(+,+)

printf/3
printf(Alias_or_stream,Format,ArgList)
printf(+,+,+)

printf_opt/3
printf_opt(Format,ArgList,Options)
printf_opt(+,+,+)

printf/4
printf(Alias_or_stream,Format,ArgList,Options)
printf(+,+,+,+)

The printf/[...] goup of predicates provides a powerful formatted printing facility closely related to the corresponding facilities in the C programming language. `printf/[...]` accepts a format string together with a list of arguments to print,

possibly a stream to print to, and possibly options. The format string contains characters to be printed, characters to control the formats of the items being printed, and argument placeholders. Figure 13 illustrates the general structure of the `printf` predicate.

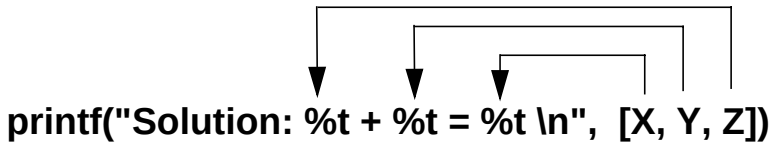


Figure 13. `printf` formatting

As a simple example, the following clause defines a predicate for adding two numbers and printing the result:

```
add(X,Y) :-
  Z is X + Y,
  printf("Solution: %t + %t = %t\n",[X,Y,Z]).
```

The `%t` placeholder tells `printf/2` that it should take the next argument from the argument list, and print it as a Prolog term. The `\n` at the end of the format string causes a newline character to be printed. The following shows the result of calling `add/2`:

add(7, 8) → **Solution: 7 + 8 = 15**

The same effect could have been obtained without `printf/2`. It is instructive to see how it is done:

```
add(X,Y) :-
  Z is X + Y,
  write(X),
  write(' + '),
  write(Y),
  write(' = '),
  write(Z),
  nl.
```

The second version of the predicate is longer and somewhat harder to read. If no arguments must be supplied to `printf` (i.e., the string contains no placeholder characters), the unary version, `printf/1`, can be used.

The fundamental formatted output predicate is `printf/4`. The first four predicates above are convenience predicates and can be defined as follows:

```
printf(Format) :-
  current_output(Stream),
  printf(Stream,Format,[],[]).

printf(Format,ArgList) :-
  current_output(Stream),
  printf(Stream,Format,ArgList,[]).

printf_opt(Format,ArgList,Options) :-
  current_output(Stream),
  printf(Stream,Format,ArgList,Options).

printf(Alias_or_stream,Format,ArgList) :-
  printf(Alias_or_stream,Format,ArgList,[]).
```

`printf/4` is closely related to the C language printf function. Roughly, the formats supported by `printf/4` are the same as those allowed by the C language printf, with the inclusion of several additional combinations, In particular, '`%t`', which indicates that the corresponding Prolog term should be output at that point. And where one would call the C language function in the form

```
printf(Format, A1, A2, ..., An),
```

one calls the Prolog `printf/2` in the form

```
printf(Format, [A1,A2,...,An]).
```

More precisely, formats are specified as follows. An ***extent expression*** consist of either a sequence of digits, or of two sequences of digits separated by a period(.); in addition, an extent expression may be prefixed with a minus sign (-). If K is an extent expression, an ***active format element*** is one of the following expressions:

`%t` `%p` `%K`s `%K`d `%K`e `%K`f `%K`g

The last five elements in this list are also called ***C format elements.*** A ***printf format*** is a quoted string, ie., and atom. (Using double quoted strings for formats is accepted for bacwards compatibility; however, it is much more wasteful of storage.) Any printf format contains zero or more active format elements, together with other text (possibly none).

The behavior of printf/[..] for the format elements `%Ks, %Kd, %Ke, %Kf,` and `%Kg` is completely in accord with the C printf, since in these cases, the argument (appropriately converted) is simply passed to the C printf. As noted above, for `%t`, the argument is printed on the output stream as a Prolog term. Finally, the format allows the programmer to take control of the formatting process as follows. Suppose that the format is `%p`, that `Stream` is the stream argument to `printf/4,` and that PArg is the argument corresponding to `%p.` Then the action of `printf/4` is determined by:

```
(PArg = Stream^PrintGoal0 ->
       call(PrintGoal0)
       ;
       (PArg = [Stream, Options]^PrintGoal0 ->
            call(PrintGoal0)
            ;
            call(PArg)
       )
).
```

`printf/4` is effectively defined as follows:

If Format = [],

printf(Alias_or_stream, Format, ArgList, Options)

succeeds; otherwise,

printf(Alias_or_stream, Format, ArgList, Options)

holds provided that:

If

Format = ["%t" | FormatTail] and ArgList = [T | ArgListTail],

then

print_term(Alias_or_stream, T, Options) and
printf(Alias_or_stream,FormatTail,ArgListTail,Options)

```
else if
    Format = ["%p" | FormatTail] & ArgList = [T | ArgListTail],
then   if
         T = S^PG & S=Alias_or_stream
       then call(PG)
       else   if
                 T = [S,O]^PG & S=Alias_or_stream & O=Options
            then call(PG)
            else
               call(T) and
               printf(Alias_or_stream,FormatTail,ArgListTail,Options)
else if Format = ["%%" | FormatTail] ,
then
    output the character % to Alias_or_stream and
    printf(Alias_or_stream,FormatTail,ArgListTail,Options),
else if
    Format = [Head | FormatTail] & Head is a C active format string
    & ArgList = [T | ArgListTail],
then
    output T to Alias_or_stream in format Head in the manner of  C printf,
    & printf(Alias_or_stream,FormatTail,ArgListTail,Options)
else
    Format = [C | FormatTail] &
    put_char(Alias_or_stream, C) &
    printf(Alias_or_stream,FormatTail,ArgListTail,Options)
```

sprintf/3
sprintf(Alias_or_stream,Format,ArgList)
sprintf(+,+,+)

bufwrite/2
bufwrite(String,Term)
bufwrite(String,Term)

bufwriteq/2
bufwriteq(String,Term)
bufwriteq(String,Term)

These very useful convenience predicates are defined by

```
sprintf(Output,Format,Args)
  :-
  open(string(Output),write,Stream),
  printf(Stream,Format,Args),
  close(Stream).

bufwrite(String,Term) :-
  open(string(String),write,Stream),
  write_term(Stream,Term,
       [line_length(10000),quoted(false),
         maxdepth(20000), quoted_strings(false)]),
  close(Stream).

bufwriteq(String,Term) :-
  open(string(String),write,Stream),
  write_term(Stream,Term,
       [line_length(10000), quoted(true),
        maxdepth(20000), quoted_strings(false)]),
  close(Stream).
```

15.8 Operator Declarations

op/3
op(Priority, Op_specifier, Operator)
op(+, +, +)

`op/3` is used to specify Operator as a syntactic operator (for the Prolog parser) according to the specifications of Priority and Op_specifier.

current_op/3

current_op(Priority, Op_specifier, Operator)
current_op(?, ?, ?)

`current_op(Priority, Op_specifier, Operator)` is true iff `Operator` is an operator with properties defined by specifier `Op_specifier` and precedence `Priority`.

Examples

currentop(P, xfy, OP). Succeeds three times if the predefined operators have not been altered, producing the following bindings:

P → 1100 OP → ;

P → 1050 OP → ->

P → 1000 OP → ,

The order in which the solutions are produced is implementation dependent.

15.9 DEC10-Style I/O Predicates

The full details of the definitions of the DEC10-style I/O predicates are presented in the file *sio_d10.pro* which is in the *alsdir* subdirectory. This file should be loaded whenever one wishes to use the DEC10- style I/O system (apart from simple calls to read/1 and write/1). Note that the predicates listed here as DEC10-style predicates have been added to the ALS Library, and so the file *sio_d10.pro* is automatically loaded by the development environment whenever one of them is called.

Below, we present conceptual definitions (which simply suppress some of the detail) of the DEC10 predicates.

see/1.

```
see(Alias_or_stream) :-
  'is input stream'(Alias_or_stream,Stream),
  !,
  set_current_input(Stream).
```

```
see(FileName) :-
  open(FileName,read,[alias(FileName)],Stream),
  set_input(Stream).
```

seeing/1.

```
seeing(Alias) :-
  current_input(Stream),
  current_alias(Alias,Stream), !.

seeing(Stream) :- current_input(Stream).
```

seen/0.

```
seen :-
  current_input(Stream),
  close(Stream).
```

tell/1.

```
tell(Alias_or_stream) :-
  'is output stream'(Alias_or_stream,Stream), !,
  set_current_output(Stream).

tell(FileName) :-
  open(FileName,write,[alias(FileName)],Stream),
  set_current_output(Stream).
```

telling/1.

```
telling(Alias) :-
  current_output(Stream),
  current_alias(Alias,Stream), !.

telling(Stream) :- current_output(Stream).
```

told/0.

```
told :-
  current_output(Stream),
  close(Stream).
```

get0/1.

```
get0(Byte) :-
  current_input(Stream),
  get_code(Stream,Byte), !.
```

get/1.

```
get(Byte) :-
  get0(Byte0),
  get_more(Byte0,Byte).

get_more(Byte0,Byte) :-
  Byte0 =< 0' ', !,
  get0(Byte1),
  get_more(Byte1,Byte).

get_more(Byte,Byte).
```

skip/1.

```
skip(Byte) :-
  get0(Byte0),
  skip_more(Byte,Byte0).

skip_more(Byte,Byte0) :-!.

skip_more(Byte,_) :-
  get0(Byte0),
  skip_more(Byte,Byte0).
```

put/1.

```
put(Byte) :-
  current_output(Stream),
  put_code(Stream,Byte), !.
```

tab/1.

```
tab(N) :-
  N =< 0, !.
```

```
tab(N) :- NN is N-1,
  put(0' '),
  tab(NN).
```

ttyflush/0.

```
ttyflush :- flush_output.
```

display/1.

```
display(X) :-
  write_term(X,[quoted(false),ignore_ops(true),numbe
  rvars(true)]).
```

15.10The user file

The file *user* represents both the keyboard (stream `user_input`) and the display (stream `user_output`). The system automatically opens stream `user_input` to be ***stdin***, and opens stream `user_output` to be ***stdout***. .

The above information is useful if you use operating system shell commands to change the standard input and output to be other than the keyboard and display. This will work properly if ALS Prolog was invoked with the `-g` and `-b` command line switches, and if the operating system supports such redirection (e.g., the Macintosh does not).

The *user* file (`user_input, user_output`) cannot be closed with `close/1`. However, you can signal end-of-file from the console on various operating systems as follows:

- On UNIX: **Control-D**
- On DOS and Win32: **Control-Z** followed by a return
- On the Macintosh: **Control-D**, or **Control-Z**.

16 Prolog Builtins: Non-I/O

16.1 Term Manipulation

16.1.1 Comparison predicates

@< /2
@> /2
@=< /2
@>= /2
== /2
\== /2

compare/3
compare(Relation, TermL, TermR)
compare(?, +, +)

@< /2, @> /2, @=< /2, @>= /2 perform comparisons on terms according to the standard order for Prolog terms. ==/2 and \==/2 perform limited identity (isomorphism) checks on their arguments. compare/3 subsumes both these comparison and identify checks.

16.1.2 Term Classification

atom/1
atomic/1
float/1
integer/1
number/1

These predicates classify terms according to the types expressed by their names.

16.1.3 Term Analysis & Synthesis

functor/3

functor(Term, Atom, Integer)
functor(?,?,?)

If `Term` is instatiated to a compound term (including an atom, which is viewed as a compound term of arity 0), then `Atom` and `Integer` are unified respectively with the functor of `Term` and the number of arguments of `Term`. Conversely, if `Atom` is an atom and `Integer` is a non-negative integer, then `Term` is unified with a compound term with functor `Atom`, and which has `Integer` number of arguments, all of which are uninstantiated variables.

arg/3
arg(Integer, Structure, Term)
arg(+, +, ?)

If `Structure` is a compound term of arity n, and `Integer` is an integer ≤ n, then `Term` is unified with the nth argument of `Structure`.

=../2
Term =.. List
? =.. ?

`Term =.. List` (pronouced 'univ') translates between terms and lists of their components. If `Term` is instantiated, List will be unified with a list of the form `[F, A1,...,An]`, where `F` is the functor of `Term` and $A_1, \ldots, A_n$ are the arguments of `Term`. Conversely, if List is of the form `[F, A1,...,An]` where `F` is an atom, then `Term` will be unified with the term whose functor is `F` and whose arguments are $A_1, \ldots, A_n$.

mangle/3
mangle(N, Structure, Term)
mangle(+,+,+)

mangle/3, though related to arg/3, destructively updates the Nth argument of Structure to become Term.

var/1
var(Term)
var(+)

nonvar/1
nonvar(Term)
nonvar(+)

`var(Term)` succeeds iff `Term` is an uninstantiated variable, and `nonvar/1` behaves exactly opposite.

16.1.4 List manipultation predicates

append/3
append(LeftList,RightList,ResultList)
append(?,?,?)

dappend/3
dappend(LeftList,RightList,ResultList)
dappend(?,?,?)

dappend/3 is a determinate version of append/3.

member/2
member(Item, List)
member(?, ?)
dmember/2
dmember(Item, List)
dmember(?, ?)

dmember/2 is a determinate version of member/2.

reverse/2
reverse(List, RevList)
reverse(?, ?)
dreverse/2,
dreverse(List, RevList)
dreverse(?, ?)

dreverse /2 is a determinate version of reverse/2.

length/2
length(List, Length)

length(+, -)

length(List,Length) causes to be unified with the number elements of List.

sort/2
sort(List, SortedList)
sort(+, -)
keysort/2
keysort(List, SortedList)
keysort(+, -)

sort/2 sorts `List` according to the standard order, merging dentical elements as defined by `==/2`, and unifying the result with SortedList. `keysort/2` expects `List` to be a list of terms of the form: `Key-Data`, sorting each pair by `Key` alone. See also the ALS Library.

16.1.5 Term Database

recorda/3
recorda(Key,Term,Ref)
recorda(Key,Term,Ref)

recordz/3
recordz(Key,Term,Ref)
recordz(Key,Term,Ref)

recorded/3
recorded(Key,Term,Ref)
recorded(Key,Term,Ref)

16.2 Atom and UIA Manipulation

atom_length/2
atom_length(Atom,Length)
atom_length(+, -)

Determines the length of an atom.

atom_concat/3
atom_concat(Atom1,Atom2,Atom)
atom_concat(?, ?, ?)

Concatenates two atoms to form a third..

sub_atom/4
sub_atom(Atom,Start,Length,SubAtom)
sub_atom(+, ?, ?, ?)

Dissects atoms. When instatiated, `Start` and `Length` must be non-negative integers, and `SubAtom` must be an atom. Can be used to extract a `SubAtom` extending `Length` chars from `Start`, and to determine if a given candidate `SubAtom` occurs in `Atom`, etc.

'$uia_alloc'/2 and relatives provide an extensive collection of routines for allocating and manipulating UIAs. They are introduced in the Chapter: Working with Uninterned Atoms.

gensym/2
gensym(Prefix, Symbol)
gensym(+, -)

Creates families of unique symbols.

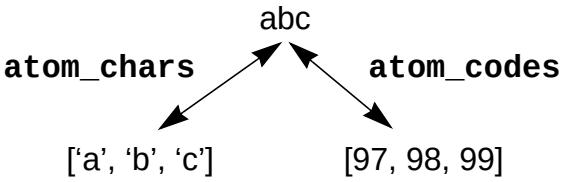
16.3 Type Conversion

atom_chars/2
atom_chars(Atom,CharList)
atom_chars(?,?)

atom_codes/2
atom_codes(Atom,CodeList)
atom_codes(?,?)

These predicates convert between atoms on the one hand, and on the other hand, ei-

ther lists of (atomic) characters, or lists of ascii character codes (Prolog strings):



number_chars/2
number_chars(Number,CharList)
number_chars(Number,CharList)

number_codes/2
number_codes(Number,CodeList)
number_codes(Number,CodeList)

In a manner exactly analogous to `atom_chars(codes)` above, these predicates convert between numbers, and lists of atomic characters or lists of ascii character codes.

term_chars/2
term_chars(Term,CharList)
term_chars(Term,CharList)

term_codes/2
term_codes(Term,CodeList)
term_codes(Term,CodeList)

In a manner exactly analogous to `atom_chars(codes)` above, these predicates convert between terms, and lists of atomic characters or lists of ascii character codes.

name/2
name(Constant,PrintName)
name(?,?)

`name/2` converts between constants and Prolog strings (lists of character codes). It is included primarily for backwards compatibility with older versions of Prolog; use of `[atom/number]_[chars/codes]` above is recommended.

16.4 Collectives

bagof/3
bagof(Template,Goal,Collection)
bagof(+, +, ?)

setof/3
setof(Template,Goal,Collection)
setof(+, +, ?)

findall/3
findall(Template,Goal,Collection)
findall(+, +, ?)

Methods of obtaining all solutions to Goal. Fail when there are no solutions.

bagOf/3
bagOf(Template,Goal,Collection)
bagOf(+, +, ?)

setOf/3
setOf(Template,Goal,Collection)
setOf(+, +, ?)

Like `bagof/3` and `setof/3`, respectively, but succeed when no solutions to `Goal` exist, unifying `Collection` with the empty list `[]`.

b_findall/4
b_findall(Template,Goal,Collection,Bound)
b_findall(+, +, -, +)

Like findall/3, except that it locates at most integer Bound > 0 number of solutions.

16.5 Prolog Database

assert/1
assert(Clause)

assert(+)

asserta/1
asserta(Clause)
asserta(+)

assertz/1
assertz(Clause)
assertz(+)

These predicates add a `Clause` to the Prolog database. If the principal functor and arity of `Clause` is `P/N`, then:

- `asserta/1` adds `Clause` before all previous clauses for `P/N`;
- `assertz/1` adds `Clause` after all previous clauses for `P/N`;
- `assert/1` adds `Clause` in some implementation-dependent position relative to all previous clauses for `P/N`.

clause/2
clause(Head, Body)
clause(+, ?)

Used to retrieve clauses `(A :- B)` [or, facts `A`] from the database where `A` unifies with `Head` and `B` unifies with `Body [true].`

retract/1
retract(Clause)
retract(+)

The current module is searched for a clause that will unify with `Clause`. The first such matching clause, if any, is removed from the database.

asserta/2
asserta(Clause, Ref)
asserta(+, -)

asserta/2
asserta(Clause, Ref)

asserta(+, -)

assertz/2
assertz(Clause, Ref)
assertz(+, -)

clause/3
clause(Head, Body, Ref)
clause(+, ?,?)

retract/2
retract(Clause, Ref)
retract(+, ?)

These predicates. all similar to their counter-parts above, add an extra argument `Ref` to the previous arguments, where Ref is an implementation-dependent ***database reference.*** A database reference obtained from `assert[a/z]/3` can be passed to `clause/3` to retrieve a clause, and to `retract/2` to delete a clause.

abolish/2
abolish(Name, Arity)
abolish(+, +)

All the clauses for the specified procedure `Name/Arity` in the current module are removed from the database

erase/1
erase(DBRef)
erase(+)

If `DBRef` is a database reference to an existing clause, `erase(DBRef)` removes that clause.

instance/2

$clauseinfo/3

$firstargkey/2

16.6 Global Variables

gv_alloc/1, make_gv/1 and relatives provide methods for manipulating global variables. See Chapter 13 (*Global Variables, Destructive Update & Hash Tables*) for a discussion.

16.7 Control

cut(!)
comma(,)
arrow (->)
semicolon (;)

abort/0
abort

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

breakhandler/0
breakhandler

call/1
call(Goal)
call(+)

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

:/2
Module:Goal
+ : +

Like call/1, but invokes Goal in the defining module Module. See Chapter 10 (*Modules*) .

catch/2
catch(Goal,Pattern,ExceptionGoal)
catch(+,+,+)

throw/0
throw(Reason)
throw(+)

These predicates provide a controlled abort mechanism, as well as access to the exception mechanism. They are introduced in Chapter 14.2 (*Exceptions.*)

fail/0

true/0

not/1
not(Goal)
not(+)

\+/1
\+(Goal)
\+(+)

`not/1` and `\+/1` are synonymous and implement negation by failure. If the `Goal` fails, then `not(Goal)` succeeds. If `Goal` succeeds, then `not(Goal)` fails.

repeat/0
repeat

`repeat/0` always succeeds, even during backtracking.

$findterm/5
'$findterm'(Functor,Arity,HeapPos,Term,NewHeapPos)
'$findterm'(+, +, +, ?, -)

A low-level predicate for searching the heap.

forcePrologInterrupt/0
callWithDelayedInterrupt/[1,2]
setPrologInterrupt/1
getPrologInterrupt/1

The predicates provide access to the ALS Prolog interrupt mechanism. See Chap-

ter 14.3 (*Interrupts.*) .

16.8 Arithmetic

See is/2 in the Reference Manual.

16.9 Program and System Management

als_system/1
als_system(InfoList)
als_system(-)

sys_env/2
sys_env(OS, Processor)
sys_env(+, +)

Predicates for obtain system environmental information.

command_line/1
command_line(Switches)
command_line(+)

Provides access to the command line by which an ALS Prolog process was invoked (including packaged applications).

compile_time/0
compile_time

Controls compile-time vs load-time execution of `:-` goals in files.

consult/1
consult(File)
consult(+)

reconsult/1
reconsult(File)
reconsult(+)

consultq/1

consultq(File)
consultq(+)

consult_to/1
consult_to(File)
consult_to(+)

consultq_to/1
consultq_to(File)
consultq_to(+)

consuld/1
consultd(File)
consultd(+)

reconsultd/1
reconsultd(File)
reconsultd+)

Various ways of dynamically loading a `File` of Prolog clauses into a running ALS Prolog program. All versions are defined in the builtins file *blt_io.pro.*

consultmessage/1
consultmessage(OnOff)
consultmessage(+)

`consultmessage(on/off)` controls whether or not messages are printed when files are consulted.

curmod/1
curmod(Module)
curmod(-)

modules/2
modules(Module,Uselist)
modules(+, -)

Provides access to information concerning modules.

gc/0
gc
Manually invokes garbage collection/compaction.

halt/0
halt
hide/1
index_proc/3

listing/[0,1]
listing
listing(Form)
listing(+)
Provides source-codes listings of clauses in the current Prolog database.

statistics/[0,2]
statistics
statistics(runtime,X)
statistics(runtime,+)
Obtain system statistics at runtime.

system/1
system(Command)
system(+)
Issue a command to the OS command processor, when supported.

module_closure/[2,3]
:- module_closure(Name,Arity,Procedure).
:- module_closure(Name,Arity).

procedures/4
all_procedures/4
all_ntbl_entries/4
Retrieve information concerning all Prolog- or C-defined procedures.

'$procinfo'/5
'$nextproc'/3
'$exported_proc'/3
'$resolve_module'/4

Retrieve detailed information about a given procedure.

16.10Date and Time

These predicates provide access to the date and time functions of the underlying operating system. They are designed to be portable across operating systems. As such, they utilize Prolog-oriented, os-independent formats for date and time. Dates are internally represented by terms of the form

```
YY/MM/DD
```

where `YY, MM, DD` are integers representing, respectively, the year, the month (counted from 1 to 12) and the day (couunted from 1 to 31, as appropriate to the month). The format of dates can be controlled by the predicate `set_date_pattern/1`. Any permuation of `YY`, `MM`, and `DD` is permitted. The predicates are defined in the builtins file *fs_cmn.pro* together with the various system-specific files *fs_unix.pro, fs_dos.pro, fs_mac.pro.*

date/1.
date(Date)
date(-)

`date/1` returns the current date in the format set by `set_date_pattern/1`.

date_pattern/4.
date_pattern(YY,MM,DD,DatePattern).
date_pattern(+,+,+,-).

This predicate consists of a single fact which provides the mapping between the three integers representing the date and the pattern expressing the date; this fact is governed by `set_date_pattern/1`.

set_date_pattern/1.

set_date_pattern(Pattern).
set_date_pattern(+).

The acceptable arguments to `set_date_pattern/1` are ground terms built up out of the ***atoms*** `yy`, `mm`, and `dd`, separated by the slash '/', such as

`mm/dd/yy` or `dd/mm/yy.`

The action of `set_date_pattern/1` is to remove the existing `date_pattern/4` fact, and to install a new fact which implements the date pattern corresponding to the input argument.

date_less/2
date_less(Date0, Date1)
date_less(+, +)

If `Date0` and `Date1` are date terms of the form `YY/MM/DD`, this predicate succeeds if and only `Date0` represents a date earlier than `Date1`.

time/1.
time(Time)
time(-)

This predicate returns a term `Time` representing the current time; Time is of the form

`HH:MM:SS`

where `HH`, `MM`, `SS` are integers in the appropriate ranges.

time_less/2.
time_less(Time0, Time1)
time_less(+, +)

If `Time0` and `Time1` are terms of the form `HH:MM:SS` representing times, this predicate succeeds if and only if `Time0` is a time earlier than `Time1`.

16.11File Names

File names and paths are one of the unpleasant ways in which operating systems differ. The file name and path predicates described in this section provide a substntial

degree of portability across operating systems. They do not claim to handle or support all possible names or path descriptions in each supported operating system. But they do deal with most normal file and path names encountered in practice. Consequently they make it possible to write fairly machine-independent code. The approach is to simply parse incoming path expressions into elementary lists, and to 'pretty-print' outgoing lists into the appropriate path expressions. The internal representation are simply lists consisting of the significant elements of the path and file name. The predicates discussed in this section are defined in the builtins file *filepath.pro.*

The primary predicates described in this Section are the following:

filePlusExt/3	- builds or decomposes a file plus extension
pathPlusFile/3	- builds or decomposes a path plus a file name
subPath/2	- builds or decomposes a subdirectory path
rootPlusPath/3	- builds or decomposes a root (disk) plus a path
rootPathFile/4	- builds or decomposes a root(disk), path, and file name
pathPlusFilesList/3	- attaches a path to each of a list of file names
same_path/2	-determines whether two file paths are the same
same_disk/2	-determines whether two disks are the same

filePlusExt/3
filePlusExt(FileName,Ext,FullName)
filePlusExt(+,+,-)
filePlusExt(-,-,+)

1) If `FileName` and `Ext` are atoms or UIAs, composes them into the complete name `FullName` with appropriate separator, e.g.,

```
foo,bar --> 'foo.bar'  .
```

2) If `FullName` is instantiated to an atom or UIA which is an appropriate complete file name, decomposes it into the `FileName` proper and the extension, `Ext`; (e.g., 'foo.bar' --> foo, bar). If `FullName` does not have an extension (e.g., 'foo'), fails.

pathPlusFile/3.
pathPlusFile(Path,File,CompletePath)

pathPlusFile(+,+,-)
pathPlusFile(-,-,+)

1)If `Path` is an atom or UIA instantiated to a path to a (sub)directory, and `File` is an atom or UIA denoting a file, composes them into a full name of the file, `CompletePath`, as for example:

```
'/u/prolog/alsdir', 'builtins.pro' -->
       '/u/prolog/alsdir/builtins.pro'
```

2)If `CompletePath` is an atom or UIA instatitated to the complete name of a file, decomposes it into the path and file parts:

```
'/u/prolog/alsdir/builtins.pro' -->
       '/u/prolog/alsdir', 'builtins.pro' ).
```

subPath/2.
subPath(PathList,SubPath)
subPath(+,-)
subPath(-,+)

1)If `PathList` is a list of atoms or UIAs which are directory names (intended to be successive subdirectories), creates the corresponding UIA SubPath denoting that path:

```
['',usr,bin,prolog] --> '/usr/bin/prolog'  .
```

2)If `SubPath` is an atom or UIA appropriately describing a (sub)path, decomposes this into a list, `PathList`, of the atoms constituting the (sub)directories in the path

```
'/usr/bin/prolog' --> ['',usr,bin,prolog] .
```

rootPlusPath/3
rootPlusPath(Disk, Path, DiskPlusPath)
rootPlusPath(+, +, -)
rootPlusPath(-, -, +)

1)If `Disk` is an atom or UIA denoting a root, and if `Path` is a list of atoms or UIAs denoting a (sub)directory path (as appropriate for subPath/2), composes these together to produce an atom `DiskPlusPath` denoting the rooted path, as for exam-

ple

```
c,[usr,bin,prolog] --> 'c:\usr\bin\prolog')
```

recognizes " (two single quotes, the empty string) as a distinguished "disk" name for systems such as unix where named disks are normally not used in path names:

```
’’, [usr,bin,prolog] --> '\usr\bin\prolog’ .
```

2)If `DiskPlusPath` is an atom or UIA denoting a rooted path (ie, beginning with a root), decomposes this to produce an atom `Disk` naming the root, and a list `Path` of atoms denoting the sequence of subdirectories in the path as appropriate for `subPath/2`:

```
'foo:\usr\bin\prolog' -->  c,[usr,bin,prolog] .
 '\usr\bin\prolog' -->  ’’,[usr,bin,prolog] .
```

rootPathFile/4
rootPathFile(Disk,Path,File,CompletePath)
rootPathFile(+,+,+,-)
rootPathFile(-,-,-,+)

1)If `Disk` and `File` are atoms or UIAs, denoting a disk and a file, respectively, and if `Path` is a list of atoms denoting a (sub)directory path, composes these to produce `CompletePath`, an atom denoting the complete name of the file:

```
c, [usr,bin,prolog], 'alspro.exe' -->
 'c:\usr\bin\prolog\alspro.exe'
```

2)If CompletePath is an atom or UIA denoting a file, decomposes this to `Disk,Path,File`:

```
 'c:\usr\bin\prolog\alspro.exe'  -->
           c, [usr,bin,prolog], 'alspro.exe'
'c:\usr\bin\prolog\foobar'  --> c, [usr,bin,prolog],
  foobar
 '/usr/bin/prolog/alspro.exe'  --> root,
  [usr,bin,prolog],
'alspro.exe'  '/usr/bin/prolog/foobar'   -->
           root, [usr,bin,prolog], foobar
```

pathPlusFilesList/3.
pathPlusFilesList(SourceFilesList, Path, ExtendedFilesList)
pathPlusFilesList(+, +, -)

If `SourceFilesList` is list of items denoting files,and if `Path` denotes a path, creates a list of atoms which consist of the `Path` prepended to each of the file names.

same_path/2
same_path(Path1, Path2)
same_path(+, +)

If `Path1` and `Path2` are two lists denoting file paths, determines whether they denote the same path, allowing for identification of uppercase and lowercase names as appropriate for the OS.

same_disk/2
same_disk(Disk1, Disk2)
same_disk(+, +)

If `Disk1` and `Disk2` are atoms denoting disks, determines whether they are the same, allowing for identification of upper and lower case letters, as appropriate for the os.

16.12File System[1]

The most important aspects of access to the file system are described in Chapter 11 on Prolog I/O. However, there are a number of further useful operations which are described in this Section. The predicates discussed in this section are defined in the builtins file *fs_cmn.pro* together with the various system-specific files *fs_unix.pro, fs_dos.pro, fs_mac.pro.* The primary predicates are the following:

Manipulating directories and files:

get_cwd/1	- returns the current working directory
change_cwd/1	- change the current working directory

1. The predicates in this Section are defined in the builtins files *fsunix.pro, fsdos.pro, fsmac.pro,* etc.

make_subdir/1	- creates a subdirectory in the current working directory
remove_subdir/1	- removes a subdirectory from the current working directory
remove_file/1	- removes a file from the current working directory
exists_file/1	- determines whether or not a file exists
exists_subdir/1	- determines whether or not a subdirectory exists
file_status/2	- returns status information concerning a file
file_size/2	- returns the size of a file

Lists of files in subdirectories:

files/2	- returns a list of files, matching a pattern, in the current directory
files/3	- returns a list of files, matching a pattern, residing in a directory
subdirs/1	- returns the list of subdirectories of the current directory
subdirs_red/1	- returns the list of subdirectories of the current directory, sans '.', '..'
collect_files/3	- returns a list of files meeting conditions
directory/3	- returns a list of files of a given type and matching a pattern

Manipulating Drives:

get_drive_status/2	- returns the status of a given drive
get_current_drive/1	- returns the current drive
get_num_logical_drives/1	- returns the number of logical drives
change_current_drive/1	- changes the current drive

change_cwd/1
change_cwd(NewDir)
change_cwd(+)

Changes the current working directory being used by the program to become `NewDir` (which must be an atom). Under DOS, this does not change the drive.

get_cwd/1
get_cwd(Path)
get_cwd(-)

Returns the current working directory being used by the program as a quoted atom. Under DOS, the drive is included.

make_subdir/1
make_subdir(NewDir)
make_subdir(+)

If `NewDir` is an atom, creates a subdirectory named `NewDir` in the current working directory, if possible.

remove_subdir/1
remove_subdir(SubDir)
remove_subdir(+)

If `SubDir` is an atom, remove the subdirectory named `SubDir` from the current working directory, if it exists.

remove_file/1
remove_file(FileName)
remove_file(+)

If `FileName` is an atom (possibly quoted) naming a file in the current working directory, removes that file.

files/2
files(Pattern,FileList)
files(+,-)

Returns the list (`FileList`) of all ordinary files in the current directory which match `Pattern`, which can include the usual '*' and '?' wildcard characters.

files/3
files(Directory, Pattern,FileList)
files(+,+,-)

Returns the list (`FileList`) of all ordinary files in the directory `Directory` which match `Pattern`, which can include the usual '*' and '?' wildcard characters.

subdirs/1
subdirs(SubdirList)
subdirs(-)

Returns the list of all subdirectories of the current working directory.

subdirs_red/1
subdirs_red(SubdirList)
subdirs_red(-)

Returns the list of all subdirectories of the current working directory, omitting '.' and '..'

collect_files/3
collect_files(PatternList,FileType,FileList)
collect_files(+,+,-)

If `PatternList` is a list of file name patterns, possibly including the usual wild-card characters '*' and '?', and if `FileType` is a standard file type (see below), then `FileList` is a sorted list of all files in the current working directory which are of type `FileType`, and which match at least one of the patterns on `PatternList`. Operates by recursively working down `PatternList` and calling `directory/3` on each element, and then doing a sorted merge on the resulting lists.

exists_file/1.
exists_file(File)
exists_file(+)

Determines whether a file exists in the current directory.

exists_subdir/1
exists_subdir(File)
exists_subdir(+)

Determines whether a subdirectory exists in the current directory.

File Types and Status

Every OS classifies files into different types and provides them with various statuses. ALS Prolog provides a (least common multiple) abstract type for files, together with the ability to utilize OS-specific types, as described in the followin table. On OSs for which a given type does not exist, requests for files of such types simply

fail, and of course, they are never returned.

Table 6:

Abstract Type	Unix Type	DOS Type	Mac Type
directory	1	16	
character_special	2		
block_special	3		
regular	4	32	
symbolic_link	5		
socket	6		
fifo_pipe	7		
unknown	0	0	
read_only		2	
hidden		4	
system		8	

Where applicable, permissions for files can be queried and manipulated. Permissions are lists of atoms representing the permission details. The following table presents the possible permissions (order is unimportant). On some systems, some subattributes such as 'execute' are meaningless.

[]
[execute]
[write]
[write,execute]
[read]

[read,execute]
[read,write]
[read,write,execute]

file_status/2
file_status(FileName, Status)
file_status(+, -)

If `FileName` is an atom naming a file in the current directory, returns a list `Status` of equations of the form

```
Tag = Value
```

which provide information on the status of the file. The four equations included on the list are:

```
type=FileType
permissions=Permissions
mod_time=ModTime
size=ByteSize
```

where `FileType` and `Permissions` are as described above. On systems where meaningful, `ModTime` is the time of last modification, or else the creation time, while `ByteSize` is the size of the file in bytes.

directory/3
directory(Pattern,FileType,List)
directory(+,+,-)

If `Pattern` is a file name pattern, including possibly the '*' and '?' wildcard characters, and if `FileType` is a numeric (internal) file type or a symbolic (abstract) file type, `directory/3` unifies `List` with a sorted list of atoms of names of files of type `FileType`, matching `Pattern`, and found in the current directory.

file_size/2
file_size(FileName,Size)
file_size(+,-)

If `File` is an atom (possibly quoted) which is the name of a file in the current working directory, `Size` is the size of that file in bytes.

get_current_drive/1
get_current_drive(Drive)
get_current_drive(-)

Returns the current logical drive. On Unix, returns the erzataz drive ''.

get_num_logical_drives/1
get_num_logical_drives(Num)
get_num_logical_drives(+)

Returns the number `Num` of logical drives. On Unix, `Num = 1`.

change_current_drive/1
change_current_drive(Drive)
change_current_drive(+)

If `Drive` is an atom describing a logical drive which exists, changes the current drive to become `Drive`. On Unix, simply succeeds with no side effects.

get_drive_status/2
get_drive_status(Drive,Status)
get_drive_status(+,-)

If `Drive` is an atom describing a logical drive which exists, returns a descriptor `Status` which describes the status of `Drive`. On Unix, `Status = 0`.

16.13I-Code Calls

$icode/4
$icode(ServiceNumber,Arg1,Arity,Arg2)

The builtin `$icode` is used to call the internal code generation function. The form of the call is

```
$icode(ServiceNumber,Arg1,Arity,Arg2)
```

When `ServiceNumber` is non-negative, it represents an abstract machine in-

struction to put in the instruction buffer. Negative `ServiceNumber` values are interpreted as commands. The remaining arguments are service dependent and should be filled in with zeros when not applicable.

init_codebuffer (-1)	Resets the internal code buffer pointer to point to the beginning of the code buffer.
name_clause (-2)	Attaches a predicate name and arity to the code currently in the icode buffer. Arg1 is a symbol (or token number) of the predicate. Arity should be set to the desired arity.
math_start (-3)	Indicates the start of an inline math computation. This command should precede the emission of a `math_begin` instruction and causes the current buffer position to be stored for use in the relative address computation at `math_end`.
math_rbranch (-4)	This should precede an rbranch instruction. It is used for the relative address calculation associated with a `math_endbranch` command.
math_end (-5)	Fills in the relative address associated with the `math_begin` instruction (which was immediately preceded by a `math_start` command).
math_reset (-6)	Causes the internal buffer pointer to be reset to the point at which `math_start` was called. This is used internally to throw away some inline math code after the compiler has decided that it can't compile it (as in '`X is 2.3`' for example).
math_endbranch (-7)	Fills in the relative branch associated with an rbranch instruction which was immediately preceded by a `math_rbranch` command.

export (-8)	Exports the predicate designated by Arg1/Arity in the *current* module.
new_module (-9)	Creates/opens a (new) module whose name is given by Arg1. If the module does not already exist, it is created and `use` declarations to user and builtins are added to the module. In addition, the current module will become the new module. This means that assert commands will place clauses in this module and predicate references within clauses will be to this module so it is desirable to call `new_module before` asserting a clause. If the module already exists, the `current module` is simply set to the module whose name is given by Arg1.
end_module (-10)	Closes the current module and sets the current module to `user`.
change_module (-11)	Changes the current module to the module whose name is given by Arg1 without creating the default use declarations.
add_use (-12)	Adds a use declaration to the module given by Arg1 to the current module.
asserta (-13)	Allocates code space and inserts the code in the icode buffer at the beginning of the predicate. The predicate should first have been named by `name_clause`. Any first argument indexing that exists for the predicate will be thrown away.
assertz (-14)	Allocates code space and appends the code in the code buffer to the end of the predicate. The predicate should first have been named with `name_clause`. Any first argument indexing

that exists for the predicate will be thrown away.

exec_query (-15) — Causes the code in the icode buffer to be executed as a query (Meaning, Answers will be displayed and yes or no will be printed.)

exec_command (-16) — Causes the code in the icode buffer to be executed as a command. Nothing will be printed regardless of success of failure.

set_cutneeded (-17) — Sets/resets the internal `cut_needed` flag. If the clause has any cuts, comma, semicolons or calls, but is not classified as a cut macro, this flag should be set. It will be set when

```
Arg1 != -1 && !Arg2
```

While this is rather arcane, there are good reasons for it internally. For consistent results, the cut-needed flag should be set for each clause sometime before asserting it. If the `cut_needed` flag is set for either assertz or asserta, an instruction to move the current choice point to the cut point will be inserted prior to creation of the first choice point.

reset_obp (-18) — Erases the the icode parameters in the .obp file back to the most recent `init_codebuffer`.

index_all (-19) — Causes indexing to be generated for all predicates. This is normally done after a consult or reconsult operation. Assert and retract operations, however, cause the indexing to be discarded, so this service may be called to redo indexing after the database has been changed via assert or retract.

index_single (-20)	not implemented
addto_autouse (-21)	Causes a module name to be added to the list of modules to be automatically used. By default, only the builtins module is automatically used by all other modules. Argument one should be the name of the module to add to the autouse list.
addto_autoname (-22)	Causes a procedure name/arity to be added to the autoname list. This is a list of procedures for which "stubs" are created when a module is initialized. By default, `call/1`, `','/2`, `';'/2`, are on this list. These stubs must exist for context dependent procedures such as `call` or `setof` to work properly. Arg1 should be set to the procedure name and `Arity` should be set to the arity.
cremodclosure (-23)	Creates a module closure . Procedures such as `asserta/1` and `bagof/3` are defined in `builtins` and yet need to know which module invoked them. The solution is to create a $n+1$ argument version of these procedures in the builtins.pro file (or elsewhere) and create a module closure. This module closure will link together the three argument version with the four argument version, installing the calling module in the fourth argument. `Arg1` should be the name of the n argument procedure. `Arity` should be n. `Arg2` should be the name of the $n+1$ argument procedure to execute after installing the module name in the $(n+1)$th argument.
hideuserproc (-24)	Used to hide user defined procedures. The first service argument (i.e., the second argument of `$icode/4`) is the name of the procedure to hide, the second is the arity of the procedure, the last is

the name of the module in which the procedure is defined. The following query will hide `user:p/0`.

```
?- $icode(-24, p, 0, user).
```

relinkdatabase (-25) — Relinks the entire database. Relinking of the program is done automatically by Version 1.1 after each consult or reconsult. However, it may still be desirable to relink the program before certain calls to assert or abolish.

Icode calls and .obp files

A `.obp` file simply consists of parameters to icode calls (along with symbol table information). During the execution of a command, it is not always desirable to keep the command in the .obp file. A simple example of this is in the DCG expander where `expand/2` is called from the parser as a command. `expand/2` will transform the DCG rule and assert it into the database. This assert operation will cause the code to be asserted in the database in addition to being added to the `.obp` file. If the expand command were retained, the assert operation would be done twice. Note also that when the `.obp` version of the file is loaded, the `expand` predicate will not be called. Only the `assert` operations that the `expand` predicate created will be performed.

Icode Instructions

The non-negative icode service numbers cause WAM instructions to be installed in the icode buffer. In the current version, argument/temporary (Ai, Xn) registers may range from 1 thru 16. Permanent variable numbers (Yn and MaxYn and EnvSize) may range from 1 thru 62. Only arities 0 thru 15 are permitted. Specifying procedure names, functors, and symbols (ProcName, Functor, Sym) is accomplished by passing in the symbol or token number if known. Integers are signed 16-bit quantities. Because of the restriction on the size of structures, NVoids should be at most 15

```
% p.
```

```
assert_p :-
   $icode(-1,0,0,0),            % initialize icode
buffer
   $icode(1,0,0,0),             % proceed
   $icode(-17,-1,0,0),        % no need for cut_btoc
                                % instruction
   $icode(-2,p,0,0),           % want to assert into
p/0.
   $icode(-14,0,0,0).           % assert the clause
% p(x).
assert_px :-
   $icode(-1,0,0,0),            % initialize icode
buffer
   $icode(25,x,0,1),            % get_symbol
   $icode(1,0,0,0),             % proceed
   $icode(-17,-1,0,0),        % no need for cut_btoc
                                % instruction
   $icode(-2,p,1,0),           % want to assert into
p/1.
   $icode(-14,0,0,0).           % assert the clause
% p(f(x),9).
assert_pfx9 :-
   $icode(-1,0,0,0),            % initialize icode
buffer
   $icode(28,f,1,1),          % get_structure f/1,A1
   $icode(44,x,0,0),            % unify_symbol  x
   $icode(26,9,0,2),          % get_integer   9,A2
   $icode(1,0,0,0),             % proceed
  $icode(-2,p,2,0),         % want to assert into p/2
  $icode(-17,-1,0,0),        % no need for a cut_btoc
                                % instruction
   $icode(-14,0,0,0).          % assertz the clause
% succ(X,Y) :- Y is X+1.
assert_succ :-
   $icode(-1,0,0,0),            % initialize icode
```

```
buffer
   $icode(-3,0,0,0),          % save buffer position
for AFP
   $icode(54,0,0,0),             % math_begin
   $icode(50,1,0,0),             % push_integerA1
   $icode(52,1,0,0),             % push_integer1
   $icode(56,0,0,0),             % add
   $icode(53,1,0,0),             % pop_integerA1
   $icode(23,1,0,2),             % get_valueX1, A2
   $icode(1,0,0,0),              % proceed
   $icode(-5,0,0,0),             % fill in relative
address for
                                 % math_begin
   $icode(32,1,0,3),             % put_valueA1,A3
   $icode(32,2,0,1),             % put_valueA2,A1
  $icode(37,'+',2,2),          % put_structure'+'/2,A2
   $icode(47,3,0,0),          % unify_local_value A3
   $icode(45,1,0,0),             % unify_integer1
   $icode(3,is,2,0),             % execute is/2
   $icode(-2,succ,2,0),          % succ/2 is the
procedure name
   $icode(-17,-1,0,0),        % reset the cut_needed
flag
   $icode(-14,0,0,0).            % assert it
```

17 The ALS Library Mechanism

The ALS Library mechanism provides a sophisticated device for managing large libraries of code in an efficient and flexible manner. Many files of potentially useful code can be available to a program without the cost of loading these files at the time the program is initially loaded. Only if program execution leads to a need for code from a particular library file is that file in fact loaded. Thereafter, execution proceeds as if the file had already been loaded.. The library mechanism is essentially invisibile to the programmer, except for a possible momentary pause when a particuular group of library predicates is first loaded. Consequently, the line between the predicates which are called 'builtin' and those which are called 'library' is quite gray.

By its nature, the library is almost always under construction. Check the contents of the *...alsdir/library/* directory for new additions.

17.1 Overview of ALS Library Mechanism and Tools.

Normally, the units making up the library are various sized (small to large) files containing code defining certain useful predicates, or defining whole subsystems of a large program.. Some of the predicates in such a file will be exported. These are the predicates which are regarded as ***library predicates***, and it is a call on one of them which must cause the library file to be loaded.

Like most symbolic languages, ALS Prolog utilizes a ***name table*** which is a hash table recording the association between names of predicates and the internal addresses at which their executable code is stored. Quite simply, the ALS library mechanism replaces the normal name table entry for the library predicates by a special 'stub' name table entry which accomplishes three things:

- it indicates that the predicate in question is a library predicate;
- it indicates the file in which the library predicate resides;
- it issues an internal ALS Prolog interrupt which is regarded as a ***library interrupt***.

In essence, execution is interrupted before execution of the called predicate, say p,

has actually commenced. During handling of the interrupt, the indicated file is loaded (really, reconsulted, which is important), and the interrupt is released, resuming normal execution at the call to p. However, since the library file reconsulted during the interrupt contains a definition of p, the special name table entry for p has been replaced by a normal name table entry p, so that execution proceeds as if the code for p had always been loaded. Note that there is no interpretive overhead for this mechanism. The sole cost is born by the predicates which are stored as library predicates. And the overhead for the library predicates is not measurably greater that their own portion of the loading time at program initialiation, were they to be loaded with the rest of the the system.

The primitive mechanisms which implement this approach to libraries are to be found in the builtins file *blt_sys.pro.* The acutal loading mechanism is defined by `load_lib/2`. The related predicate `force_libload_all/2` can be used to force the loading a list of library files. This can be useful during construction of a stand-alone package. The low level mechanism for installing a library-type name table entry is `libhide/3.` The ALS Prolog system uses the file blt_lib.pro to record information about files which are to be treated as library files. This allows great flexibility, and in particular allows users and developers to add their own packages as library files. A tool for managing this process is described in the ***ALS Development Tools Guide***.

The collection of library predicates is steadily developing. The library includes such facilities as the macro processing tools and the structure definition/abstracton tools. These are described in their own sections of this manual or the ALS Tools Guide. The survey below lists the remaining groups which have been installed as of the date of writing of this chapter.

17.2 Lists: Algebraic List Predicates (listutl1.pro)

append/2
append(ListOfLists, Result)
append(+, -)

-- appends a list of lists together

If ListOfLists if a list, each of whose elements is a list, Result is obtained by appending the members of ListOfLists together in order.

intersect/2
intersect(L,IntsectL)
intersect(+,-)
-- returns the intersection of a list of lists

If L is a list of lists, returns the intersection IntsectL of all the list appearing on L.

intersect/3
intersect(A,B,AintB)
intersect(+,+,-)
-- returns the intersection of two lists

If A and B are lists, returns the intersection AintB of A and B, which is the collection of all items common to both lists.

list_diff/3
list_diff(A, B, A_NotB)
list_diff(+, +, +)
-- returns the ordered difference of two lists

If A and B are lists, returns the difference A-B consisting of all items on A, but not on B.

list_diffs/4
list_diffs(A,B,A_NotB,B_NotA)
list_diffs(+,+,-,-)
-- returns both ordered differences of two lists

If A and B are lists, returns both the difference A-B together with the difference B-A.

sorted_merge/2
sorted_merge(ListOfLists, Union)
sorted_merge(+, -)
-- returns the sorted union of a list of lists

If ListOfLists is a list of lists, Union is the sorted merge (non-repetitive union) of the members of ListsOfLists.

sorted_merge/3
sorted_merge(List1, List2, Union)
sorted_merge(+, +, -)
-- returns the sorted union of two lists

If List1 and List2 are lists of items, Union is the sorted merge (non-repetitive union) of List1 and List2.

symmetric_diff/3
symmetric_diff(A,B,A_symd_B)
symmetric_diff(+,+,-)
-- returns the symmetric difference of two lists

If A and B are lists, returns the symmetric difference of A and B, which is the union of A-B and B-A.

union/3
union(A,B, AuB)
union(+,+, -)
-- returns the ordered union of two lists

If A and B are lists, returns the ordered union of A and B, consisting of all items occurring on either A or B, with all occurrences of items from A occurring before any items from B-A; equivalent to:

```
append(A,B-A,AuB);
```

If both lists have the property that each element occurs no more than once, then the union also has this property.

17.3 Lists: Positional List Predicates (listutl2.pro)

at_most_n/3
at_most_n(List, N, Head)
at_most_n(+, +, -)
-- returns initial segment of list of length =< N

If List is a list and N is a non-negative integer, Head is the longest initial segment of List with length =< N.

change_nth/3
change_nth(N, List, NewItem)
change_nth(+, +, +)
-- destructively changes the Nth element of a list

If N is a non-negative integer, List isa list, and NewItem is any non-var object, destructively changes the Nth element of List to become NewItem; this predicate numbers the list beginning with 0.

deleteNth/3
deleteNth(N, List, Remainder)
deleteNth(+, +, -)
-- deletes the Nth element of a list

If N is a non-negative integer and List is a list, then Remainder is the result of deleting the Nth element of List; this predicate numbers the list beginning with 1.

get_list_tail/3
get_list_tail(List, Item, Tail)
get_list_tail(+, +, -)
-- returns the tail of a list determined by an element

If List is a list and Item is any object, Tail is the portion of List extending from the leftmost occurrence of Item in List to the end of List; fails if Item does not belong to List.

list_delete/3
list_delete(List, Item, ResultList)
list_delete(+, +, -)
-- deletes all occurrences of an item from a list

If List is a list, and Item is any object, ResultList is obtained by deleting all occurrences of Item from List.

nth/3
nth(N, List, X)
nth(+, +, -)
-- returns the nth element of a list

If List is a list and N is a non-negative integer, then X is the nth element of List.

nth_tail/4
nth_tail(N, List, Head, Tail)
nth_tail(+, +, -, -)
-- returns the nth head and tail of a list

If List is a list and N is a non-negative integer, then Head is the portion of List up to but not including the Nth element, and tail is the portion of List from the Nth element to the end.

position/3
position(List, Item, N)
position(+, +, -)
-- returns the position number of an item in a list

If List is a list and Item occurs in List, N is the number of the leftmost occurrence of Item in List; fails if Item does not occur in List.

position/4
position(List, Item, M, N)
position(+, +, +, -)
-- returns the position number of an item in a list

If List is a list and Item occurs in List, N-M is the number of the leftmost occurrence of Item in List; fails if Item does not occur in List.

sublist/4
sublist(List,Start,Length,Result)
sublist(+,+,+,-)
-- extracts a sublist from a list

If List is an arbitrary list, Result is the sublist of length Length beginning at position Start in List.

subst_nth/4
subst_nth(N, List, NewItem, NewList)
subst_nth(+, +, +, -)
-- non-destructively changes the Nth element of a list

If N is a non-negative integer, List is list, and NewItem is any non-var object, NewList is the result of non-destrctively changing the Nth element of List to become |NewItem; this predicate numbers the list beginning with 0.

17.4 Lists: Miscellaneous List Predicates (listutl3.pro)

check_default/4
check_default(PList, Tag, Default, Value)
check_default(+, +, +, -)
-- *looks up an equation on a list, with default*

PList is a list of equations of the form tag = value check_default(PList, Tag, Default, Value) succeeds if Tag=Value belongs to PList; otherwise, if Default=Value

encode_list/3
encode_list(Items, Codes, CodedItems)
encode_list(+, +, -)
-- *combines a list of items with a list of codes*

If Items and Codes are lists of arbitrary terms of the same length, then CodedItems is the list of corresponding pairs of the form Code-Item

flatten/2
flatten(List, FlatList)
flatten(+, -)
-- *flattens a nested list*

If List is a list, some of whose elements may be nested lists, FlatList is the flattened version of List obtained by traversing the tree defining List in depth-first, left-to-right order; compound structures other than list structures are not flattened.

merge_plists/3
merge_plists(LeftEqnList, RightEqnList, MergedLists)
merge_plists(+, +, -).
-- *(recursively) merges two tagged equation lists*

LeftEqnList and RightEqnList are lists of equations of the form tag = value MergedLists consists of all equations occurring in either LeftEqnList or RightEqn-

List, where if the equations Tag=LVal and Tag = RVal occur in LeftEqnList and RightEqnList, respectively, MergedLists will contain the equation Tag = MVal where: a)If both of LVal and RVal are lists, then MVal is obtained by recursively calling merge_plists(LVal, RVal, MVal); b)Otherwise, MVal is LVal.

n_of/3
n_of(N, Item, Result)
n_of(+, +, -)
-- creates a list of N copies of an item

Result is a list of length N all of whose elements are the entity Item.

nobind_member/2
nobind_member(X, List)
nobind_member(+, +)
-- tests list membership without binding any variables

nobind_member(X, List) holds and only if X is a member of List; if the test is successful, no variables in either input are bound.

number_list/2
number_list(List, NumberedList)
number_list(+, -)
-- creates a numbered list from a source list

If List is a list, NumberedList is a list of terms of the form N-Item, where the Item components are simply the elements of List in order, and N is a integer, sequentially numbered the elements of List.

number_list/3
number_list(Items, StartNum, NumberedItems)
number_list(+, +, -)
-- numbers the elements of a list

If Items is a list, and StartNum is an integer, NumberedItems is the list obtained by replacing each element X in Items by N-X, where N is the number of the position of X in Items.

output_prolog_list/1

output_prolog_list(List)
output_prolog_list(+)

-- outputs items on a list, one to a line

Outputs (to the current output stream) each item on List, one item to a line, followed by a period.

remove_tagged/3
remove_tagged(EqnList, TagsToRemove, ReducedEqnList)
remove_tagged(+, +, -).

-- removes tagged equations from a list

EqnList is a list of equations of the form tag = value and TagsToRemove is a list of atoms which are candidates to occur as tags in these equations. ReducedEqnList is the result of removing all equations beginning with a tag from TagsToRemove from the list EqnList.

struct_lookup_subst/4
struct_lookup_subst(OrderedTags, DefArgs, ArgSpecs, ArgsList)
struct_lookup_subst(+, +, +, -)

-- performs substs for structs package constructors

OrderedTags and DefArgs are lists of the same length; so will be ArgsList. ArgSpecs is a list of equations of the form Tag = Value where each of the Tags in such an equation must be on the list OrderedTags (but not all OrderedTags elements must occur on ArgSpecs); in fact, ArgSpecs can be empty. The elements X of ArgsList are defined as follows: if X corresponds to Tag on OrderedTags, then: if Tag=Val occurs on ArgSpecs, X is Val; otherwise, X is the element of DefArgs corresponding to Tag.

17.5 Tree Predicates (avl.pro)

avl_create/1
avl_create(Tree)
avl_create(-)

-- create an empty tree.

avl_create(Tree) creates an empty avl tree which is unified with Tree.

avl_inorder/2
avl_inorder(Tree,List)
avl_inorder(+,-)
-- returns list of keys in an avl tree in in-order traversal

If Tree is an avl tree, List is the ordered list of keys encountered during an inorder traversal of Tree.

avl_inorder_wdata/2
avl_inorder_wdata(Tree,List)
avl_inorder_wdata(+,-)
-- returns list of keys and data in an avl tree in in-order traversal

If Tree is an avl tree, List is the ordered list of terms of the form Key-Data encountered during an inorder traversal of Tree.

avl_insert/4
avl_insert(Key,Data,InTree,OutTree)
avl_insert(+,+,+,-)
-- inserts a node in an avl tree

Inserts Key and Data into the avl-tree passed in through InTree giving a tree which is unified with OutTree. If the Key is already present in the tree, then Data replaces the old data value in the tree.

avl_search/3
avl_search(Key,Data,Tree)
avl_search(+,?,+)
-- searches for a key in an avl tree

Tree is searched in for Key. Data is unified with the corresponding data value if found. If Key is not found, avl_search will fail.

17.6 Miscellaneous Predicates (commal.pro)

flatten_comma_list/2
flatten_comma_list(SourceList, ResultList)
flatten_comma_list(+, -)

-- *flattens nested comma lists and removes extraneous trues'*

If SourceList is a comma list (i.e., (a,b,c,...)), then ResultList is also a comma list which is the result of removing all extraneous nesting and all extraneous occurrences of true'.'

17.7 I/O Predicates (iolayer.pro)

17.8 Control Predicates (lib_ctl.pro)

bagOf/3
bagOf(Pattern, Goal, Result)
bagOf(+, +, -)

-- *Like bagof/3, but succeeds with empty list on no solutions*

bagOf/3 is just like bagof/3, except that if Goal has no solutions, bagof/3 fails, whereas bagOf/3 will succeed, binding Result to [].

max/3
max(A,B,M)
max(+,+,-)

-- *computes the maximum of two numbers*

If A and B are ground expressions which evaluate to numbers under is/2, the M will be their maximum value.

min/3
min(A,B,M)
min(+,+,-)

-- *computes the minimum of two numbers*

If A and B are ground expressions which evaluate to numbers under is/2, the M will be their minimum value.

setOf/3
setOf(Pattern, Goal, Result)
setOf(+, +, -)

-- *Like setof/3, but succeeds with empty list on no solutions*

setOf/3 is just like setof/3, except that if Goal has no solutions, setof/3 fails, whereas setOf/3 will succeed, binding Result to [].

17.9 Prolog Database Predicates (misc_db.pro)

assert_all/1
assert_all(ClauseList)
assert_all(+)
-- asserts each clause on ClauseList in the current module

If ClauseList is a list of clauses, asserts each of these clauses in the current module.

assert_all0/2
assert_all0(ClauseList,Module)
assert_all0(+,+)
-- asserts each clause on ClauseList in module Module

If ClauseList is a list of clauses, asserts each of these clauses in module Module.

assert_all_refs/3
assert_all_refs(Module,ClauseList, RefsList)
assert_all_refs(+,+, -)
-- asserts a list of clauses, in a module, returning a list of refs

If Module is a module, and if ClauseList is a list of terms which can be asserted as clauses, then assert_all_refs/3 causes each term on ClauseList to be asserted in Module, and returns RefsList as the list of corresponding references to these asserted clauses.

erase_all/1
erase_all(RefsList)
erase_all(+)
-- erases each clauses referenced by a list of clause references

If RefsList is a list of clauses references, causes each clause corresponding to one of these references to be erased.

17.10I/O Predicates (misc_io.pro)

colwrite/4
colwrite(AtomList,ColPosList,CurPos,Stream)
colwrite(+,+,+,+)
-- writes atoms in AtomList at column positions in ColPosList

If AtomList is a list of atoms (symbols or UIAs), and if ColPosList is a list of monotonically increasing positive integers of the same length as AtomList, and if CurPos is a positive integer (normally 1), and Stream is valid output stream (in text mode), this predicate outputs the items on AtomList to Stream, starting each element of AtomList at the postition indicated by the corresponding element of ColPosList. If a given item would overflow its column, it is truncated. Normally, CurPos = 1 and ColPosList begins with an integer greater than 1, so that the first column position is implicit.

copyFiles/2
copyFiles(SourceFilesList, TargetSubDirPath)
copyFiles(+, +)
-- copies files to a directory

If SourceFilesList is a list of file names, and TargetSubDirPath is either an atom or an internal form naming a directory, copies all of the indicated files to files with the same names in TargetSubDirPath.

gen_file_header/[3,4]
gen_file_header(OutStream,SourceFile,TargetFile)
gen_file_header(OutStream,SourceFile,TargetFile,ExtraCall)
gen_file_header(+,+,+)
gen_file_header(+,+,+,+)
-- output a header suitable for a generated file

OutStream is a write stream, normally to file TargetFile. Given SourceFile = fooin, and TargetFile = foooout, gen_file_header/3 outputs a header of the following format on OutStream:

```
/*==============================================
          fooout
```

```
          --Generated from: fooin
          Date: 94/4/17   Time: 9:52:53
 *===============================================*/
```

In gen_file_header/4, the argument ExtraCall is called just before the printing of the lower comment line. Thus, if ExtraCall were

```
printf(OutStream,'         -- by zipper_foo\n',[]),
```

the output would look like:

```
/*===============================================
          fooout
          --Generated from: fooin
          Date: 94/4/17   Time: 9:52:53
          -- by zipper_foo
*================================================*/
```

putc_n_of/3
putc_n_of(Num, Char, Stream)
putc_n_of(+,+,+)

-- output Num copies of the char with code Char to Stream

Num should be a positive integer, and Char should be the code of a valid character; Stream should be an output stream in text mode. Outputs, to Stream, Num copies of the characater with code Char.

read_terms/1
read_terms(Term_List)
read_terms(-)

-- reads a list of Prolog terms from the default input stream

Reads a list (Term_List) of all terms which can be read from the default input stream

read_terms/2
read_terms(Stream,Term_List)
read_terms(+,-)

-- reads a list of Prolog terms from stream Stream

Reads a list (Term_List) of all terms which can be read from the stream Stream.

read_lines/[1,2]
read_lines(Stream,Line_List)
read_lines(+,-)

Reads a list (Line_List) of all lines which can be read from the stream Stream.

17.11I/O Predicates (simplio.pro)

17.12String Manipulation Predicates (strings.pro)

asplit/4
asplit(Atom,Splitter,LeftPart,RightPart)
asplit(+,+,-,-)

-- divides an atom as determined by a character

If Atom is any atom or UIA, and if Splitter is the character code of of a character, then, if the character with code Splitter occurs in Atom, LeftPart is an atom consisting of that part of Atom from the left up to and including the leftmost occurrence of the character with code Splitter, and RightPart is the atom consisting of that part of Atom extending from immediately after the end of LeftPart to the end of Atom.

asplit0/4
asplit0(AtomCs,Splitter,LeftPartCs,RightPartCs)
asplit0(+,+,-,-)

-- divides a list of character codes as determined by a character code

If AtomCs is a list of character codes, and if Splitter is the character code of of a character, then, if the character with code Splitter occurs in AtomCs, LeftPart is the list consisting of that part of AtomCs from the left up to and including the leftmost occurrence of Splitter, and RightPart is the atom consisting of that part of AtomCs extending from immediately after the end of LeftPart to the end of AtomCs.

head/4
head(Atom,Splitter,Head,Tail)
head(+,+,-,-)

-- splits an list into segments determined by a character code

If Atom is a list of character codes, splits Atom into Head and tail the way asplit would, using the first occurrence of Splitter; on successive retrys, usings the succeeding occurrences of Spliter as the split point.

head0/4
head0(List,Splitter,Head,Tail)
head0(+,+,-,-)
-- *splits a character code list into segments determined by a code*

If List is a list of character codes, splits List into Head and tail the way asplit0 would, using the first occurrence of Splitter; on successive retrys, usings the succeeding occurrences of Spliter as the split point.

17.13Miscellaneous Predicates (xlists.pro)

xlist_append/2
xlist_append(ListOfXLists, Result)
xlist_append(+, -)
-- *appends together an ordinary list of extensible lists*

If ListOfXLists is an ordinary list of xtensible lists, then Result is obtained by serially xappending each of the xlists occurring on ListOfXLists.

xlist_append/3
xlist_append(Left, Right, Result)
xlist_append(+, +, -)
-- *appends two extensible lists*

xlist_head/2
xlist_head(XList, Result)
xlist_head(+, -)
-- *returns the head of an extensible list*

xlist_init/1
xlist_init(Result)
xlist_init(-)
-- *creates a freshly initialized extensible list*

xtensible lists are carriedaround in the form (Head, Tail) The actual list may be a standard extensible list [a,b,c,d | T] or may be a comma separated list: (a, (b, (c, (d, T)))) It is up to the routines using these tools to bind the tail variable to the correct structure, or to createthe correct type of structure for xappending to another such xlist.

xlist_make/3
xlist_make(Head, Tail, Result)
xlist_make(+, +, -)
-- Makes an extenstible list data structure from Head,Tail

xlist_tail/2
xlist_tail(XList, Result)
xlist_tail(+, -)
-- returns the tail of an extensible list

xlist_unit_c/2
xlist_unit_c(First, Result)
xlist_unit_c(+, -)
-- creates a freshly initialized comma-type xlist with first elt

xlist_unit_l/2
xlist_unit_l(First, Result)
xlist_unit_l(+, -)
-- creates a freshly initialized ordinary xlist with first elt