ISO Prolog:
A Summary of the Draft Proposed Standard

Michael A. Covington
Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602–7415 U.S.A.
mcovingt@ai.uga.edu

Preprint of Appendix A of:
Prolog Programming in Depth
by Michael A. Covington, Donald Nute, and André Vellino
(Prentice–Hall, in preparation)

July 1, 1993

Contents

Appendix A

Summary of Draft ISO Prolog

This appendix is a summary of the March 1993 draft ISO standard for the Prolog language, ISO/IEC JTC1 SC22 WG17 N110 ("Prolog: Part 1, General core"). As this is written (June 1993), the proposed standard is still subject to change, and standard–conforming Prolog implementations have not yet appeared.[1] Section A.9 summarizes the June 1993 proposal for a module system, ISO/IEC JTC1 SC22 WG17 N111 ("Prolog: Part 2, Modules"), which is much farther from final form than N110.

The information given here is only a sketch; anyone needing definitive details is urged to consult the ISO documents themselves.

The draft ISO standard does not include definite clause grammars (DCGs), nor the Edinburgh file–handling predicates (`see`, `seen`, `tell`, `told`, etc.). Implementors are, however, free to keep these for compatibility, and nothing that conflicts with them has been introduced.

The proposed standard does not presume that you are using the ASCII character set. The numeric codes for characters can be whatever your computer uses.

A.1 Syntax of Terms

A.1.1 Comments and Whitespace

Whitespace ("layout text") consists of blanks, end–of–line marks, and comments. (Implementations commonly treat tabs and formfeeds as equivalent to blanks.)

You can put whitespace before or after any term, operator, bracket, or argument separator, as long as you do not break up an atom or number and do not separate a functor from the opening parenthesis that introduces its argument list. Thus `f(a,b,c)` can be written `f( a , b , c )`, but there cannot be whitespace between `f` and `(`.

Whitespace is sometimes required, e.g., between two graphic tokens. For example, `* *` is two occurrences of the atom `*`, but `**` is one atom.

There are two types of comments. One type begins with `/*` and ends with `*/`; the other type begins with `%` and ends at the end of the line. Comments can be of zero length (e.g., `/**/`).

It is not possible to nest comments of the same type (for example, `/* /* */` is a complete, valid comment). But a comment of one type can occur in a comment of the other type (`/* % thus */`).

STYLE NOTE: Because nesting is not permitted, we recommend using `%` for ordinary comments and using `/* */` only to comment out sections of code.

A.1.2 Variables

A variable name begins with a capital letter or the underscore mark (`_`) and consists of letters, digits, and/or underscores. A single underscore mark denotes an anonymous variable.

[1] A rough draft of this appendix was circulated by Internet; I want to thank Jan Burse, Jo Calder, Klaus Daessler, Markus Fromherz, Fergus Henderson, Andreas Kagedal, and especially Roger Scowen for pointing out errors.

A.1.3 Atoms (Constants)

There are four types of atoms:

- A series of letters, digits, and/or underscores, beginning with a lower–case letter;
- A series of 1 or more characters from the set

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

  provided it does not begin with ‘`/*`’ (such atoms are called GRAPHIC TOKENS);
- The special atoms `[]` and `{}` (see section A.1.6 below);
- A series of arbitrary characters in single quotes.

Within single quotes, a single quote is written double (e.g., `'don''t panic'`). A backslash at the very end of the line denotes continuation to the next line, so that

```
'this is \
 an atom'
```

is equivalent to `'this is an atom'` (that is, the line break is ignored). Note however that when used this way, the backslash must be at the physical end of the line, not followed by blanks or comments. (In practice, some implementations are going to have to permit blanks because it is hard or impossible to get rid of them.)[2]

Another use of the backslash within a quoted atom is to denote special characters, as follows:

`\a`	alert character (usually the beep code, ASCII 7)
`\b`	backspace character
`\f`	formfeed character
`\n`	newline character or code (implementation dependent)
`\r`	return without newline
`\t`	(horizontal) tab character
`\v`	vertical tab character (if any)
`\x23\`	character whose code is hexadecimal 23 (likewise for any number of hex digits)
`\23\`	character whose code is octal 23 (likewise for any number of octal digits)
`\\`	backslash
`\'`	single quote
`\"`	double quote
`` \` ``	backquote

The last two of these will never be needed in a quoted atom. They are used in other types of strings that take these same escape sequences, but are delimited by double quotes or backquotes.

A.1.4 Numbers

Integers are written in any of the following ways:

- As a series of decimal digits, e.g., `012345`;
- As a series of octal digits preceded by `0o`, e.g., `0o567`;
- As a series of hexadecimal digits preceded by `0x`, e.g., `0x89ABC`;

[2] A line break written as such cannot be part of the atom; for example,

```
'this and
that'
```

is not a valid atom. Instead, use the escape sequence `\n`.

- As a series of binary digits preceded by `0b`, e.g., `0b10110101`;
- As a character preceded by `0'`, e.g., `0'a`, which denotes the numeric code for the character `a`. (The character is written exactly as if it were in single quotes; that is, if it is a single quote it must be written twice, and an escape sequence such as `\n` is treated as a single character.)

Floating–point numbers are written only in decimal. They consist of at least one digit, then (optionally) a decimal point and more digits, then (optionally) `E`, an optional plus or minus, and still more digits. For example:

```
234    2.34    2.34E5    2.34E+5    2.34E-10
```

Note that `.234` and `2.` are not valid numbers.

A minus sign can be written before any number to make it negative (e.g., `-3.4`). Notice that this minus sign is part of the number itself; hence `-3.4` is a number, not an expression.

A.1.5 Character Strings

The ISO standard provides four ways of representing character–string data:

- As atoms (`'like this'`). Unfortunately, atoms take up space in the symbol table, and some implementations limit the size of each atom, or the total number of atoms, or both. The standard itself does not recognize any such limits.
- As lists of one–character atoms (`[l,i,k,e,' ',t,h,i,s]`).
- As Edinburgh–style strings `"like this"`, where a string is a list of numeric codes (e.g., `"abc"` = `[97,98,99]`).

  Actually, `"abc"` = `[97,98,99]` only on computers that use the ASCII character codes. But the equivalence `"abc"` = `[0'a,0'b,0'c]` holds on all computers, because `0'a` represents the numeric code for `a`.
- As strings delimited by backquotes (`` `like this` ``) *if* the implementor wants to implement them. No operations are defined on this type of string, and they are not required to be implemented at all.

As you might guess, these four options reflect considerable disagreement among the standardizers. For a long time, Edinburgh strings weren't going to be included at all, but they were finally added to the March 1993 draft.

The quotes that delimit a string or atom, whichever kind they may be, are written double if they occur within the string (`'it''s'`, `"it""s"`, `` `it``s` ``). Double quoted strings and backquoted strings recognize the same backslash escape sequences as are used in quoted atoms (Section A.1.3).

Table A.1 shows all the built–in predicates that relate to character string operations. Most perform operations on atoms or lists of characters rather than lists of numeric codes.

A.1.6 Structures

The ordinary way to write a structure is to write the functor, an opening parenthesis, a series of terms separated by commas, and a closing parenthesis: `f(a,b,c)`. We call this FUNCTOR NOTATION, and it can be used even with functors that are normally written as operators (e.g., `2+2` = `+(2,2)`).

Lists are defined as rightward–nested structures using the operator '`.`' (which is not an infix operator). For example,

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

Table A.1: Built–in predicates for character–string operations.

`atom_length(Atom,Integer)`
Length (in characters) of `Atom` is `Integer`.

`atom_concat(Atom1,Atom2,Atom3)`
Concatenating `Atom1` and `Atom2` gives `Atom3`. (Either `Atom3`, or both `Atom1` and `Atom2`, must be instantiated.)

`sub_atom(Atom,N,L,Sub)`
The substring of `Atom` beginning at the `N`th character and `L` characters long is `Sub`. (`Atom` must be instantiated.) A query such as `sub_atom(abcabc,N,L,ab)` produces multiple solutions upon backtracking.

`char_code(Char,Code)`
Relates a character (i.e., a one–character atom) to its numeric code (ASCII, or whatever the computer uses). (Either `Char` or `Code`, or both, must be instantiated.)

`atom_chars(Atom,Chars)`
Interconverts atoms with lists of the characters that represent them, e.g., `atom_chars(abc,[a,b,c])`. (Either `Atom` or `Chars`, or both, must be instantiated.)

`atom_codes(Atom,String)`
Like `atom_chars`, but uses a list of numeric codes, i.e., a string.

`number_chars(Num,Chars)`
Interconverts numbers with lists of the characters that represent them, e.g., `number_chars(23.4,['2','3','.','4'])`. (Either `Num` or `Chars`, or both, must be instantiated.)

`number_codes(Num,String)`
Like `number_chars`, but uses a list of numeric codes, i.e., a string.

These predicates raise error conditions if an argument is the wrong type. Note that `name/2` is not included in the standard.

Table A.2: Predefined operators of Draft ISO Prolog.

Priority	*Specifier*	*Operators*
1200	`xfx`	`:-  -->`
1200	`fx`	`:-  ?-`
1100	`xfy`	`;`
1050	`xfy`	`->`
1000	`xfy`	`,`
700	`xfx`	`= \= == \== @< @=< @> @>= is =:= =\= < =< > >= =..`
500	`yfx`	`+  -  /\  \/`
400	`yfx`	`*  /  //  rem  mod  <<  >>`
200	`xfx`	`**`
200	`xfy`	`^`
200	`fy`	`\  -`
100	`xfx`	`@`
50	`xfx`	`:`

Many of the standardizers want to add `\+`, presumably with priority 900 and specifier `fy`.

There can be only one `|` in a list, and no commas after it.

Curly brackets have a special syntax that is used in implementing definite clause grammars, but can also be used for other purposes. Any term enclosed in `{ }` is treated as the argument of the special functor '`{}`':

```
{one}  =  {}(one)
```

Recall that commas can act as infix operators; thus,

```
{one,two,three} = {}(',',(one,',',(two,three)))
```

and likewise for any number of terms.

The standard does not include definite clause grammars, but does include this syntactic "hook" for implementing them. You are, of course, free to use curly brackets for any other purpose.

A.1.7 Operators

The predefined operators of Draft ISO Prolog are shown in Table A.2. The meanings of the operators will be explained elsewhere in this appendix as they come up; `@` and `:` are to be used in the module system (Part 2 of the draft, not yet released). Some operators, such as `?-` and `-->`, are not given a meaning in the standard, but are preserved for compatibility reasons.

The SPECIFIER of an operator, such as `xfy`, gives both its CLASS (infix, prefix, or postfix) and its ASSOCIATIVITY. Associativity determines what happens if there are two infix operators of equal priority on either side of an argument. For example, in `2+3+4`, `3` could be an argument of either the first or the second `+`, and the associativity `yfx` specifies that the grouping on the left should be formed first, treating `2+3+4` as equivalent to `(2+3)+4`. The Prolog system parses an expression by attaching operators to their arguments, starting with the operators of the lowest priority, thus:

```
2 + 3 * 4 =:= X         (original expression)
2 + *(3,4) =:= X        (after attaching *, priority 400)
+(2,*(3,4)) =:= X       (after attaching +, priority 500)
=:=(+(2,*(3,4)),X)      (after attaching =:=, priority 700)
```

Terms that are not operators are considered to have priority 0.

The same atom can be an operator in more than one class (such as the infix and prefix minus signs). To avoid the need for unlimited lookahead when parsing, the same atom cannot be both an infix operator and a postfix operator.

A.1.8 Commas

The comma has three functions: it separates arguments of functors, it separates elements of lists, and it is an infix operator of priority 1000. Thus `(a,b)` (without a functor in front) is a structure, equivalent to `','(a,b)`.

A.1.9 Parentheses

Parentheses are allowed around any term. The effect of parentheses is to override any grouping that may otherwise be imposed by operator priorities. Operators enclosed in parentheses do not function as operators; thus `2(+)2` is a syntax error.

A.2 Program Structure

A.2.1 Programs

The draft standard does not define "programs" per se, because Prolog is not a (wholly) procedural language. Rather, the standard defines PROLOG TEXT, which consists of a series of clauses and/or directives, each followed by '`.`' and then whitespace.

The standard does not define `consult` or `reconsult`; instead, the mechanism for loading and querying a Prolog text is left up to the implementor.

A.2.2 Directives

The draft standard defines the following set of directives, which is somewhat tentative:

`:- dynamic(Pred/Arity).`
: The specified predicate is to be dynamic (modifiable at run time). (See also section A.9.)

`:- multifile(Pred/Arity).`
: The specified predicate can contain clauses loaded from more than one file. (The `multifile` declaration must appear in each of the files, and if the predicate is declared dynamic in any of the files, it must be declared dynamic in all of them.)

`:- discontiguous(Pred/Arity).`
: The clauses of the specified predicate are not necessarily together in the file. (If this declaration is not given, the clauses of each predicate are required to be contiguous.)

`:- op(Priority,Associativity,Atom).`
: The atom is to be treated syntactically as an operator with the specified priority and associativity (e.g., `xfy`).

CAUTION: An `op` directive in the program file affects the syntax while the program is being loaded; the standard does not require that its effect persist after the loading is complete. Traditionally, an `op` declaration permanently changes the syntax used by the Prolog system (until the end of the session), thus affecting all further `read`s, `write`s, and `consult`s; the standard permits but does not require this behavior. See also section A.9.

However, `op` can also be called as a built–in predicate while the program is running, thereby determining how `read` and `write` will behave at run time.

An operator can be deprived of its operator status by declaring it to have priority 0 (in which case its class and associativity have no effect, but must still be declared as valid values).

`:- char_conversion(Char1,Char2).`
: This specifies that if character conversion is enabled (see "Flags," Section A.5), all occurrences of `Char1` that are not in quotes should be read as `Char2`. Note that, to avoid painting yourself into a corner, you should normally put the arguments of `char_conversion` in quotes so that they won't be subject to conversion.

The situation with `char_conversion` is analogous to `op` — the standard does not require its effect to persist after the program finishes loading. However, you can also call `char_conversion` as a built–in predicate at execution time, to determine how characters will be converted at run time.

`:- initialization(Goal).`
This specifies that as soon as the program is loaded, the goal `Goal` is to be executed. There can be more than one `initialization` directive, in which case all of the goals in all of them are to be executed, in an order that is up to the implementor.

`:- include(File).`
Specifies that another file is to be read at this point exactly as if its contents were in the main file. (Apparently, a predicate split across two files using `include` does not require a `multifile` declaration, since the loading is all done at once.)

`:- ensure_loaded(File).`
Specifies that in addition to the main file, the specified file is to be loaded. If there are multiple `ensure_loaded` directives referring to the same file, it is only loaded once.

Note that *directives are not queries* — the standard does not say you can embed arbitrary queries in your program, nor that you can execute directives as queries at run time (except for `op` and `char_conversion`, which are, explicitly, also built–in predicates). Traditionally, directives have been treated as a kind of query, but the standard, with advancing compiler technology in mind, does not require them to be.

A.3 Control Structures

A.3.1 Conjunction, disjunction, `fail`, and `true`

As in virtually all Prologs, the comma (`,`) means "and," the semicolon (`;`) means "or," `fail` always fails, and `true` always succeeds with no other action.

A.3.2 Cuts

The cut (`!`) works in the traditional way. When executed, it succeeds and throws away all backtrack points between itself and its CUTPARENT. Normally, the cutparent is the query that caused execution to enter the current clause. However, if the cut is in an environment that is OPAQUE TO CUTS, the cutparent is the beginning of that environment. Examples of environments that are opaque to cuts are:

- The argument of the negation predicate (`fail_if`, traditionally written `not` or `\+`).
- The argument of `call`, which can of course be a compound goal, such as `call((this,!,that))`.
- The left–hand argument of '`->`' (see below).
- The goals that are arguments of `once`, `catch`, `findall`, `bagof`, and `setof` (and, in general, any other goals that are arguments of predicates).

A.3.3 If–then–else

The "if–then–else" construct (`Goal1 -> Goal2 ; Goal3`) tries to execute `Goal1`, and, if successful, proceeds to `Goal2`; otherwise, it proceeds to `Goal3`. The semicolon and `Goal3` can be omitted. Note that:

- Only the first solution to `Goal1` is found; any backtrack points generated while executing `Goal1` are thrown away.
- If `Goal1` succeeds, execution proceeds to `Goal2`, and then:
  - If `Goal2` fails, the whole construct fails.
  - If `Goal2` succeeds, the whole construct succeeds.

  - If `Goal2` has multiple solutions, the whole construct has multiple solutions.

- If `Goal1` fails, execution proceeds to `Goal3`, and then:
  - If `Goal3` fails, the whole construct fails.
  - If `Goal3` succeeds, the whole construct succeeds.
  - If `Goal3` has multiple solutions, the whole construct has multiple solutions.
- If `Goal1` fails and there is no `Goal3`, the whole construct fails.
- Either `Goal2` or `Goal3` will be executed, but not both (not even upon backtracking).
- If `Goal1` contains a cut, that cut only has scope over `Goal1`, not the whole clause. That is, `Goal1` is opaque to cuts.
- The whole if–then–else structure has multiple solutions if `Goal1` succeeds and `Goal2` has multiple solutions, or if `Goal1` fails and `Goal3` has multiple solutions. That is, backtrack points in `Goal2` and `Goal3` behave normally.
- Cuts in `Goal2` and `Goal3` have scope over the entire clause (i.e., behave normally).

Note that the semicolon in `Goal1 -> Goal2 ; Goal3` is not the ordinary disjunction operator; if it were, you would be able to get solutions to `Goal1 -> Goal2` and then, upon backtracking, also get solutions to `Goal3`. But this never happens. Rather, `->` and `;` have to be interpreted as a unit.

STYLE NOTE: We do not recommend mixing cuts with if–then or if–then–else structures.

A.3.4 Variable goals, `call`

Variables can be used as goals. A term `G` which is a variable occurring in place of a goal is converted to the goal `call(G)`. Note that `call` is opaque to cuts.

A.3.5 `repeat`

The predicate `repeat` works in the traditional way, i.e., whenever backtracking reaches it, execution proceeds forward again through the same clauses as if another alternative had been found.

A.3.6 `once`

The query `once(Goal)` finds exactly one solution to `Goal`. It is equivalent to `call((Goal,!))` and is opaque to cuts.

A.3.7 Negation (`fail_if`)

The negation predicate is called `fail_if` and is opaque to cuts. That is, `fail_if(Goal)` is like `call(Goal)` except that its success or failure is the opposite. Note that `fail_if` is not an operator, and that extra parentheses are required around compound goals (e.g., `fail_if((this,that))`).

As this is written (mid–1993), there is strong sentiment in favor of reintroducing `\+` into the standard as a prefix operator with the same meaning as `fail_if`.

A.4 Error Handling

A.4.1 `catch` and `throw`

The control structures `catch` and `throw` are provided for handling errors and other explicitly programmed exceptions. They make it possible to jump out of multiple levels of procedure calls in a single step.

The query `catch(Goal1,Arg,Goal2)` is like `call(Goal1)` except that if, at any stage during the execution of `Goal1`, there is a call to `throw(Arg)`, then execution will immediately jump back to the `catch` and proceed to `Goal2`. Here `Arg` can be a variable or only partly instantiated; the only requirement is that the `Arg` in the `catch` must match the one in the `throw`. Thus, `Arg` can include information to tell `catch` what happened.

In `catch`, `Goal1` and `Goal2` are opaque to cuts.

A.4.2 Errors detected by the system

When the system detects a runtime error, it executes a `throw(error(Type,Info))`, where `Type` is the type of error and `Info` contains other information that is up to the implementor.

If the user's program has executed a matching `catch`, execution jumps back to there; otherwise, the system prints out an error message and stops. Thus, you can use `catch` to catch system–detected errors, not just your own calls to `throw`.

The possible values of `Type` are:

`instantiation_error`
: An argument was uninstantiated in a place where uninstantiated arguments are not permitted.

`type_error(Type,Term)`
: An argument should have been of type `Type` (`atom`, `body` (of clause), `callable` (goal), `character`, `compound` (= structure), `constant`, `integer`, `list`, `number`, or `variable`), but `Term` is what was actually found.

`domain_error(Domain,Term)`
: Like `type_error`, except that a DOMAIN is a set of possible values, rather than a basic data type. Examples are `character_list` and `stream_or_alias`. Again, `Term` is the argument that caused the error.

`existence_error(ObjType,Term)`
: Something does not exist that is necessary for what the program is trying to do. Examples are `operator`, `procedure`, and `past_end_of_stream` (when you read past the end of a file). Here, again, `Term` is the argument that caused the error.

`permission_error(Operation,ObjType,Term)`
: The program attempted something that is not permissible (such as repositioning a non–repositionable file). `Term` and `ObjType` are as in the previous example, and `Operation` is `access_clause`, `create`, `input`, `modify`, or the like.

`representation_error(ErrorType)`
: An implementation–defined limit has been violated, for example by trying to handle `'ab'` as a single character. Values of `ErrorType` are `character`, `character_code`, `exceeded_max_arity`, and `flag`.

`calculation_error(ErrorType)`
: An arithmetic error has occurred. Types are `overflow`, `underflow`, `zero_divide`, and `undefined`.

`resource_error(Resource)`
: The system has run out of some resource (such as memory or disk space).

`syntax_error`
: The system has attempted to read a term that violates Prolog syntax. This can occur during program loading, or at run time (executing `read` or `read_term`).

`system_error`
: This is the catch–all category for other implementation–dependent errors.

For further details see the latest ISO documents.

Table A.3: Flags defined in the Draft ISO Prolog Standard.

`bounded` (`true` or `false`)
True if integer arithmetic gives erroneous results outside a particular range (as when you add $32767 + 1$ on a 16–bit computer and get -32768). False if the range of available integers is unlimited (as with Lisp "bignums").

`max_integer` (an integer)
The greatest integer on which arithmetic works correctly. Defined only if `bounded` is true.

`min_integer` (an integer)
The least integer on which arithmetic works correctly. Defined only if `bounded` is true.

`integer_rounding_function` (`down` or `toward_zero`)
The direction in which negative numbers are rounded by `//` and `rem`.

`char_conversion` (`on` or `off`)
Controls whether character conversion is enabled. Can be set by the program.

`debug` (`on` or `off`)
Controls whether the debugger is in use (if so, various predicates may behave nonstandardly). Can be set by the program.

`max_arity` (an integer or `unbounded`)
The maximum permissible arity for functors.

`undefined_predicate` (`error`, `fail`, or `warning`)
Controls what happens if an undefined predicate is called. Can be set by the program.

A.5 Flags

A FLAG is a parameter of the implementation that the program may need to know about. The built–in predicates `current_prolog_flag(Flag,Value)` and `set_prolog_flag(Flag,Value)` allow the program to obtain and, where applicable, change the values of flags.

Table A.3 lists the flags defined in the draft standard. Any specific implementation is likely to have many more.

A.6 Arithmetic

A.6.1 Where expressions are evaluated

Arithmetic expressions are evaluated in the following contexts:

- The right–hand argument of `is` (e.g., `X is 2+3`).
- Both arguments of the comparison predicates `=:=` `=\=` `<` `>` `=<` `>=`.

A.6.2 Functors allowed in expressions

The EVALUABLE FUNCTORS that are permitted in expressions are listed in Table A.4.

The arithmetic system of the draft ISO standard is based on other ISO standards for computer arithmetic; see the draft standard itself for full details. The draft Prolog standard requires all arithmetical operations to give computationally reasonable results or raise error conditions.

Table A.4: Functors that can be used in arithmetic expressions.

`N + N`	Addition
`N - N`	Subtraction
`N * N`	Multiplication
`N / N`	Floating–point division
`I // I`	Integer division
`I rem I`	Remainder
`I mod I`	Modulo
`N ** N`	Exponentiation (result is floating–point)
`-N`	Sign reversal
`abs(N)`	Absolute value
`atan(N)`	Arctangent (in radians)
`ceiling(N)`	Smallest integer not smaller than `N`
`cos(N)`	Cosine (argument in radians)
`exp(N)`	Natural antilogarithm, e^N
`sqrt(N)`	Square root
`sign(N)`	Sign (-1, 0, or 1 for negative, zero, or positive `N`)
`float(N)`	Convert to floating–point
`floor(X)`	Largest integer not greater than `X`
`log(N)`	Natural logarithm, $\log_e N$
`sin(N)`	Sine (argument in radians)
`truncate(X)`	Integer equal to the integer part of `X`
`round(X)`	Integer nearest to `X`
`I >> J`	Bit–shift `I` rightward `J` bits
`I << J`	Bit–shift `I` leftward `J` bits
`I /\ J`	Bitwise **and** function
`I \/ J`	Bitwise **or** function
`\ I`	Bitwise complement (reverse all bits of `I`)

Here `I` and `J` denote integers, `X` denotes floating–point numbers, and `N` denotes numbers of either type.

A.7 Input and Output

A.7.1 Overview

Except for `read`, `write`, `writeq`, and `nl`, the traditional Edinburgh input–output predicates are not included in the standard. Instead, a new, very versatile i–o system is proposed. Here is a simple example of file output:

```
test :- open('/usr/mcovingt/myfile.txt',write,MyStream,[type(text)]),
        write_term(MyStream,'Hello, world',[quoted(true)]),
        close(MyStream,[force(false)]).
```

Notice that each input–output operation can name a STREAM (an open file) and can give an OPTION LIST. To take the defaults, the option lists can be empty, and in some cases even omitted:

```
test :- open('/usr/mcovingt/myfile.txt',write,MyStream,[]),
        write_term(MyStream,'Hello, world',[]),
        close(MyStream).
```

A.7.2 Opening a stream

A STREAM is an open file (or other file–like object) that can be read or written sequentially. You can refer to a stream either by its HANDLE (an implementation–dependent term that gets instantiated when you open the stream) or its ALIAS (a name that you give to the stream).

By default, the streams `user_input` and `user_output` are already open, referring to the keyboard and the screen respectively, and are the current input and output streams. But current input and output can be redirected.

To open a stream, use the predicate `open(Filename,Mode,Stream,Options)`, where:

- `Filename` is an implementation–dependent file designator (normally a Prolog atom);
- `Mode` is `read`, `write`, or `append`;
- `Stream` is a variable that will be instantiated to an implementation–dependent "handle";
- `Options` is an option list, possibly empty.

The contents of the option list can include:

- `type(text)` (the default) or `type(binary)`. A text file consists of printable characters arranged into lines; a binary file contains any data whatsoever, and is read byte by byte.
- `reposition(true)` or `reposition(false)` (the default). A repositionable stream (e.g., a disk file) is one in which it is possible to skip forward or backward to specific positions.
- `alias(Atom)` to give the stream a name. For example, if you specify `alias(accounts_receivable)`, you can write `accounts_receivable` as the `Stream` argument of subsequent operations on this stream.
- A specification of what to do upon repeated attempts to read past end of file:
  - `eof_action(error)` to raise an error condition;
  - `eof_action(eof_code)` to make each attempt return the same code that the first one did (e.g., `-1` or `end_of_file`); or
  - `eof_action(reset)`, to examine the file again and see if it is now possible to read past what used to be the end (e.g., because of data written by another concurrent process).

  Rather surprisingly, the draft standard specifies no default for this option.

Implementors are free to add other options.

A.7.3 Closing a stream

The predicate `close(Stream,Options)` closes a stream. It can be written `close(Stream)` if the option list is empty.

The option list can include `force(false)` (the default) or `force(true)`; the latter of these says that if there is an error upon closing the stream (e.g., a diskette not in the drive), the system shall assume that the stream was successfully closed anyway, without raising an error condition.

A.7.4 Stream properties

The predicate `stream_property(Stream,Property)` lets you determine the properties of any currently open stream, like this:

```
?- stream_property(user_input,mode(What)).
What = read
```

Properties include the following:

- `file_name(...)`, the file name;
- `mode(M)`, where `M` is `input` or `output`;
- `alias(A)`, where `A` is the stream's alias, if any;
- `position(P)`, where `P` is an implementation–dependent term giving the current position of the stream;
- `end_of_stream(E)`, where `E` is `at`, `past`, or `no`, to indicate whether reading has just reached end of file, has gone past it, or has not reached it.
- `eof_action(A)`, where `A` is as in the options for `open`.
- `reposition(B)`, where `B` is `true` or `false` to indicate repositionability.
- `type(T)`, where `T` is `text` or `binary`.

Implementations are free to define other properties.

A.7.5 Reading and writing characters

Table A.5 summarizes the input–output predicates that deal with single characters. They are usable on both text and binary files; on a binary file, they treat bytes as characters.

The standard *does not specify whether keyboard input is buffered or unbuffered;* that is considered to be an implementation–dependent matter.

A.7.6 Reading terms

Table A.6 shows the predicates for reading terms. Each of them reads a term from a text stream; the term must be followed by a period and then by whitespace, and must conform to Prolog syntax.

A new feature in the draft standard gives you some access to the variables in the input. Traditionally, if you `read` a term with variables in it, such as `f(X,Y,X)`, then you get a term in which the relative positions of the variables are preserved, but the names are not, such as `f(_0001,_0002,_0001)`.

Now, however, by specifying the option `variable_names(List)`, you can also get a list that pairs up the variables with their names, like this:

```
?- read_term(Term,[variable_names(Vars)]).
f(X,Y,X).  (typed by user)
Term = f(_0001,_0002,_0001)
Vars = [_0001='X',_0002='Y']
```

Table A.5: Single–character input and output predicates.

`get_char(Stream,Char)`
: Reads a character (as a one–character atom). Returns `end_of_file` at end of file.

`get_char(Char)`
: Same, using the current input stream.

`get_code(Stream,Code)`
: Reads a byte as a numeric code. Returns `-1` at end of file.

`get_code(Code)`
: Same, using the current input stream.

`put_char(Stream,Char)`
: Writes `Char`, which must be a one–character atom. (Equivalent to `write(Char)`, but presumably faster.)

`put_char(Char)`
: Same, using the current output stream.

`put_code(Stream,Code)`
: Writes a byte given its numeric value.

`put_code(Code)`
: Same, using the current output stream.

Table A.6: Predicates for reading terms.

`read_term(Stream,Term,Options)`
: Reads a term from `Stream` using options in list.

`read_term(Term,Options)`
: Same, using current input stream.

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

`read(Term)`
: Like `read_term(Term,[])`.

All of these return the atom `end_of_file` at end of file.

Table A.7: Predicates for writing terms.

`write_term(Stream,Term,Options)`
: Outputs a term onto a text stream using the option list.

`write_term(Term,Options)`
: Same, using the current output stream.

`write(Stream,Term)`
: Like `write_term(Stream,Term,[numbervars(true)]`.

`write(Term)`
: Same, using the current output stream.

`write_canonical(Stream,Term)`
: Like `write_term` with the options `[quoted(true), ignore_ops(true)]`.

`write_canonical(Term)`
: Same, using current output stream.

`writeq(Stream,Term)`
: Like `write_term` with the options `[quoted(true), numbervars(true)]`.

`writeq(Term)`
: Same, using current output stream.

The import of this is that it lets you write your own user interface for the Prolog system (or any Prolog–like query processor). You can accept a query, store a list that gives the names of its variables, and then eventually print out the names alongside the values.

There are also two less elaborate options. The option `singletons(List)` gives you a list, in the same format, of just the variables that occurred only once in the term — useful if you're reading Prolog clauses and want to detect misspelled variable names. And `variables(List)` gives you a list of just the variables, without their names (such as `[_0001,_0002]`).

A.7.7 Writing terms

Table A.7 lists the predicates for writing terms. The following options are available:

- `quoted(true)` puts quotes around all atoms and functors that would require them in order to be read by `read/1`.

- `ignore_ops(true)` writes all functors in functor notation, not as operators (e.g., `+(2,2)` in place of `2+2`).

- `numbervars(true)` looks for terms of the form `'$VAR'(1)`, `'$VAR'(2)`, `'$VAR'(3)`, etc., and outputs them as `A`, `B`, etc.

  The significance of this is that `'$VAR'`–terms are often used to replace variables when there is a need to instantiate all the variables in a term. By printing the term out with this option, its variables can be made to look like variables again.

A.7.8 Other input–output predicates

Table A.8 lists some additional input–output predicates.

Table A.8: Miscellaneous input–output predicates.

`current_input(Stream)`
: Unifies `Stream` with the handle of the current input stream.

`current_output(Stream)`
: Unifies `Stream` with the handle of the current output stream.

`set_input(Stream)`
: Redirects current input to `Stream`.

`set_output(Stream)`
: Redirects current output to `Stream`.

`flush_output(Stream)`
: Causes all output that is buffered for `Stream` to actually be written.

`flush_output`
: Same, but uses current output stream.

`at_end_of_stream(Stream)`
: True if the stream is at or past end of file (i.e., the last character or term has been read). (Presumably, a `read` or `read_term` consumes all the whitespace following the term that it has read; but the standard does not make this clear.)

`set_stream_position(Stream,Pos)`
: Repositions a stream (use `stream_property` to obtain a term that represents a position).

`nl(Stream)`
: Starts a new line on `Stream` (which should be text).

`nl`
: Same, using current output stream.

`op(Priority,Specifier,Term)`
: Alters the set of operators during execution. See sections A.1.7, A.2.2.

`current_op(Priority,Specifier,Term)`
: Determines the operator definitions that are currently in effect. Any of the arguments, or none, can be instantiated. Gives multiple solutions upon backtracking as appropriate.

`char_conversion(Char1,Char2)`
: Alters the set of character conversions during execution. See sections A.2.2, A.5.

`current_char_conversion(Char1,Char2)`
: True if `char_conversion(Char1,Char2)` is in effect (see sections A.2.2, A.5). Either argument, or none, may be instantiated. Gives multiple solutions upon backtracking.

A.8 Other Built–In Predicates

This section briefly describes all the other built–in predicates described in the draft ISO standard.

A.8.1 Unification

`Arg1 = Arg2`
: Succeeds by unifying `Arg1` with `Arg2` in the normal manner (i.e., the same way as when arguments are matched in procedure calls). Results are undefined if you try to unify a term with another term that contains it (e.g., `X = f(X)`, or `f(X,g(X)) = f(Y,Y)`). (Commonly, such a situation produces cyclic pointers that cause endless loops when another procedure later tries to follow them.)

`unify_with_occurs_check(Arg1,Arg2)`
: Succeeds by unifying `Arg1` with `Arg2`, but explicitly checks whether this will attempt to unify any term with a term that contains it, and if so, fails:

```
?- unify_with_occurs_check(X,f(X)).
no
```

This version of unification is often assumed in work on the theory of logic programming.

`Arg1 \= Arg2`
: Succeeds if the two arguments cannot be unified (using the normal unification process).

A.8.2 Comparison

(See also the arithmetic comparision predicates `< =< > >= =:=` in section A.6.)

`Arg1 == Arg2`
: Succeeds if `Arg1` and `Arg2` are the same term. Does not unify them and does not attempt to instantiate variables in them.

`Arg1 \== Arg2`
: Succeeds if `Arg1` and `Arg2` are not the same term. Does not unify them and does not attempt to instantiate variables in them.

`Arg1 @< Arg2`
: Succeeds if `Arg1` precedes `Arg2` in alphabetical order. All variables precede all floating–point numbers, which precede all integers, which precede all atoms, which precede all structures. Within terms of the same type, the alphabetical order is the collating sequence used by the computer, and shorter terms precede longer ones.

`Arg1 @=< Arg2`
: Succeeds if `Arg1 @< Arg2` or `Arg1 == Arg2`. Does not perform unification or instantiate variables.

`Arg1 @> Arg2`
: Like `@<` with the order of arguments reversed.

`Arg1 @>= Arg2`
: Like `@=<` with the order of arguments reversed.

A.8.3 Type tests

`var(Arg)`
: Succeeds if `Arg` is uninstantiated.

`nonvar(Arg)`
: Succeeds if `Arg` is at least partly instantiated.

`atomic(Arg)`
: Succeeds if `Arg` is an atom or a number.

`compound(Arg)`
Succeeds if `Arg` is a compound term (a structure, including lists but not `[]`).

`atom(Arg)`
Succeeds if `Arg` is an atom.

`number(Arg)`
Succeeds if `Arg` is a number (integer or floating–point).

`integer(Arg)`
Succeeds if `Arg` is an integer. Note that this tests its data type, not its value. Thus `integer(3)` succeeds but `integer(3.0)` fails.

`real(Arg)`
Succeeds if `Arg` is a floating–point number. Thus `real(3.3)` succeeds but `real(3)` fails.

A.8.4 Creating and decomposing terms

`functor(Term,F,A)`
Succeeds if `Term` is a compound term, `F` is its functor, and `A` (an integer) is its arity; or if `Term` is an atom or number equal to `F` and `A` is zero. (Either `Term`, or both `F` and `A`, must be instantiated.) Some examples:

```
?- functor(f(a,b),F,A).
F = f
A = 2

?- functor(What,f,2).
What = f(_0001,_0002)

?- functor(What,f,0).
What = f

?- functor(What,3.1416,0).
What = 3.1416
```

`arg(N,Term,Arg)`
Succeeds if `Arg` is the `N`th argument of `Term` (counting from 1):

```
?- arg(1,f(a,b,c),What).
What = a
```

Both `N` and `Term` must be instantiated.

`Term =.. List`
Succeeds if `List` is a list consisting of the functor and all arguments of `Term`, in order. `Term` or `List`, or both, must be at least partly instantiated.

```
?- f(a,b) =.. What.
What = [f,a,b]

?- What =.. [f,a,b]
What = f(a,b)
```

`copy_term(Term1,Term2)`
Makes a copy of `Term1` replacing all occurrences of each variable with a fresh variable (like changing `f(A,B,A)` to `f(W,Z,W)`). Then unifies that copy with `Term2`.

```
?- copy_term(f(A,B,A),What).
A = _0001
B = _0002
What = f(_0003,_0004,_0003)
```

A.8.5 Manipulating the knowledge base

Note that ONLY DYNAMIC PREDICATES CAN BE MANIPULATED. Static predicates are compiled into a form that is inaccessible to some or all of the built–in predicates described here. Nonetheless, some implementations may treat static predicates as dynamic.

`clause(Head,Body)`

Succeeds if `Head` matches the head of a dynamic predicate, and `Body` matches its body. The body of a fact is considered to be `true`. `Head` must be at least partly instantiated. Thus, given

```
green(X) :- moldy(X).
green(kermit).
```

we get:

```
?- clause(green(What),Body).
What = _0001,  Body = moldy(_0001)  ;
What = kermit, Body = true
```

`current_predicate(Functor/Arity)`

Succeeds if `Functor/Arity` gives the functor and arity of a currently defined non–built–in predicate, whether static or dynamic:

```
?- current_predicate(What).
What = green/1
```

Gives multiple solutions upon backtracking.

Note that `current_predicate(Functor/Arity)` succeeds even if all the clauses of the predicate have been retracted (or if the predicate was declared dynamic but no clauses were ever asserted), but not if the predicate has been abolished.

`asserta(Clause)`

Adds `Clause` at the beginning of the clauses for its predicate. If there are no clauses for that predicate, the predicate is created and declared to be dynamic. If the predicate already has some clauses and is static, an error condition is raised.

`assertz(Clause)`

Like `asserta`, but adds the clause at the end of the other clauses for its predicate.

NOTE: `assert` (without `a` or `z`) is not included in the standard.

`retract(Clause)`

Removes from the knowledge base a dynamic clause that matches `Clause` (which must be at least partly instantiated). Gives multiple solutions upon backtracking.

Note that the fact `green(kermit)` could be retracted by any of the following queries:

```
?- retract(green(kermit)).
?- retract((green(kermit) :- true)).
?- retract((green(_) :- _)).
```

NOTE: `retractall` is not included in the standard.

`abolish(Functor/Arity)`

Completely wipes out the dynamic predicate designated by `Functor/Arity`, as if it had never existed. Its dynamic declaration is forgotten, too, and `current_predicate` no longer recognizes it.

This is a more powerful move than simply retracting all the clauses, which would leave the dynamic declaration in place and leave `current_predicate` still aware of the predicate.

A.8.6 Finding all solutions to a query

`findall(Term,Goal,List)`

Finds each solution to `Goal`; instantiates variables to `Term` to the values that they have in that solution; and adds that instantiation of `Term` to `List`. Thus, given

```
green(kermit).
green(crabgrass).
```

we get the following results:

```
?- findall(X,green(X),L).
L = [kermit,crabgrass]

?- findall(f(X),green(X),L).
L = [f(kermit),f(crabgrass)]
```

This is the simplest way to get a list of the solutions to a query. The solutions found by `findall` are given in the order in which the normal searching–and–backtracking process finds them.

`bagof(Term,Goal,List)`

Like `findall(Term,Goal,List)` except for its treatment of the FREE VARIABLES of `Goal` (those that do not occur in `Term`).

Whereas `findall` would try all possible values of all variables, `bagof` will pick the first set of values for the free variables that succeeds, and use only that set of values when finding the solutions in `List`.

Then, if you ask for an alternative solution to `bagof`, you'll get the results of trying another set of values for the free variables. An example:

```
parent(michael,cathy).
parent(melody,cathy).
parent(greg,stephanie).
parent(crystal,stephanie).

?- findall(Who,parent(Who,Child),L).
L = [michael,melody,greg,crystal]

?- bagof(Who,parent(Who,Child),L).    % Child is free variable
L = [michael,melody] ;                % with Child = cathy
L = [greg,crystal]                    % with Child = stephanie
```

If in place of `Goal` you write `Term^Goal`, any variables that occur in `Term` will not be considered free variables. Thus:

```
?- bagof(Who,Child^parent(Who,Child),L).
L = [michael,melody,greg,crystal]
```

The order of solutions obtained by `bagof` is up to the implementor.

`setof(Term,Goal,List)`

Like `bagof(Term,Goal,List)`, but the elements of `List` are sorted into alphabetical order (see `@<` under "Comparisons" above) and duplicates are removed.

A.8.7 Terminating execution

`halt`

Exits from the Prolog system (or from a compiled program).

`halt(N)`

Exits from the Prolog system (or from a compiled program), passing the integer `N` to the operating system as a return code. (The significance of the return code depends on the operating system. For example, in MS–DOS and UNIX, return code 0 is the usual way of indicating normal termination.)

```
:- module(my_list_stuff).
:- export([last/2,reverse/2]).

:- begin_module(my_list_stuff).

last([E],E).
last([_|E],Last) :- last(E,Last).

reverse(List1,List2) :- reverse_aux(List1,[],List2).

reverse_aux([H|T],Stack,Result) :-  reverse_aux(T,[H|Stack],Result).
reverse_aux([],Result,Result).

:- end_module.
```

Figure A.1: Example of a module.

A.9 Modules

A.9.1 Preventing name conflicts

Ordinarily, in a Prolog program, there cannot be two different predicates with the same name and arity. This can pose a problem when two programmers, writing different parts of the same program, inadvertently choose the same name and arity for different predicates.

The solution is to divide large programs into MODULES, or sections, each of which has its own namespace. Names defined in one module are not recognized in other modules unless explicitly made visible there. Thus, like–named predicates in different modules do not conflict.

A.9.2 Example of a module

Some Prolog vendors have had module systems for several years, but none is quite like the proposed ISO system.

In the proposed system, there are, by default, two modules, `system` (for built–in predicates) and `user` (for user–defined predicates). The predicates in `system` are visible in `user` and all other modules.

The user can create more modules ad libitum; Figure A.1 shows an example. The module consists of two parts: an INTERFACE, specifying what is to be made callable from other modules, and a BODY, giving the actual predicate definitions.

This module is named `my_list_stuff` and, crucially, `last/2` and `reverse/2` are callable from other modules but `reverse_aux/3` is not. Thus, `reverse_aux` will not conflict with anything that happens to have the same name elsewhere.

To use a predicate in one module which is defined in another, the defining module must EXPORT it and the calling module must IMPORT it. Thus, any module that wants to call `reverse` (as defined here) must import `my_list_stuff`.

Note that importing a module is not the same thing as loading it into memory (using `compile`, `consult`, or the like). In order to have access to a module, you must do both.

A.9.3 Module syntax

Basically, exporting is done in the module interface, while defining and importing are done in the module body. The syntax is:

:- module(*name*).

Various `export`, `reexport`, and `meta` directives

`:- end_module.` (optional if `begin_module` is the next directive)

`:- begin_module(` *name* `).`

Predicate definitions and `import`, `meta`, `op`, and `dynamic` declarations

`:- end_module.`

Here `op` and `dynamic` work the same way as if you aren't using the module system, except that they have scope only over one module. The other declarations work as follows:

`:- export([Pred/Arity,Pred/Arity...]).`
: The listed predicates (defined in this module) are made callable from other modules that import them. (Used in interface.)

`:- import(Module).`
: All the predicates that exported by `Module` are to be imported into (and hence usable in) the current module. (Used in module body.)

`:- import(Module,[Pred/Arity,Pred/Arity...]).`
: Same, but only the specified predicates are imported.

`:- reexport(Module).`
: Combination of `import` and `export`. All the predicates that are exported by `Module` are imported into the current module and are also exported by it. (Used in interface.)

`:- reexport(Module,[Pred/Arity,Pred/Arity...]).`
: Same, but only the specified predicates are reexported.

`:- meta([Pred/Arity,Pred/Arity...]).`
: The specified predicates are defined to be METAPREDICATES (see next section). This declaration can be used in either the interface or the body. If it is used in the interface, the predicates are also exported.

A.9.4 Metapredicates

A METAPREDICATE is a predicate that needs to know what module it is called from. Built–in metapredicates include `abolish`, `asserta`, `assertz`, `clause`, `current_predicate`, `current_visible`, and `retract`, all of which manipulate the predicates in the module that they are called from (not the module they are defined in); and `bagof`, `setof`, `findall`, `catch`, `call`, and `once`, all of which take goals as arguments and need to be able to execute them in the module they are called from.

Users can also define their own metapredicates by declaring them `meta` (see previous section). Within a metapredicate, the goal `calling_context(Module)` will retrieve the name of the module from which the predicate was called.

A.9.5 Explicit module qualifiers

If, instead of `Goal`, you write `Module:Goal`, you gain the ability to call any predicate of `Module`, whether or not that module has exported it or the current module has imported it. In the example in Figure A.1, the query

```
?- my_list_stuff:reverse_aux([a,b,c],X,Y).
```

would work from any module, even though `reverse_aux` is not exported by the module that defines it.

By writing `Goal@Module`, you can specify what module a metapredicate should *think* it was called from. For example,

```
?- current_predicate(What) @ my_module.
```

would retrieve the predicates that are defined in `my_module` rather than in the current module. (Note that `current_predicate` is *defined in* `system`; you would be making it think it was *executing in* `my_module`.)

A.9.6 Additional built–in predicates

`calling_context(Module)`
: Instantiates its argument to the module from which the current clause was called. For use in metapredicates.

`current_module(Module)`
: Succeeds if its argument is the name of any currently existing module. Arguments need not be instantiated.

`current_visible(Pred/Arity,Module)`
: Succeeds if `Pred/Arity` describes a predicate that is defined in `Module` and is visible (callable) from the module in which this query is taking place. Arguments need not be instantiated.

A.9.7 A word of caution

The module system is much farther from final form than the rest of the proposed ISO standard. Substantial changes are still quite possible.