IC Prolog][

Version 0.96, for Sun Workstations
28 September 1993
Department of Computing, Imperial College, London

by Yannis Cosmadopoulos and Damian Chu

1 Overview

1.1 Introduction

ICP is an Edinburgh style prolog based on the WAM. In addition it includes the following special features.

`threads` ICP allows multiple threads in a single process. This allows the solving of multiple goals in pseudo parallel, achieved through the use of pre-emptive time-slicing.

`parlog` ICP includes a parlog engine. A parlog thread can be started and facilities are provided for parlog to call prolog and vice versa.

`Logic & Objects`
Logic & Objects is a object-oriented layer on top of Prolog.

`TCP` A TCP interface allows for the communication between processes over a communication network.

`Mailboxes`
Mailbox is a high-level communication model that is simple and yet has many more features than TCP.

`user-defined error handling`
Users may define their own procedures for handling and recovering from errors.

`user-defined file types`
Users may define their own file types and then supply hooks for reading and transforming terms in those files. This is a generalisation of term expansion.

Source files are assumed to have the suffix "`.pl`" while compiled files have the extension "`.icp`". Built in predicates dealing with these files will automatically append these extensions to files if they are not present.

1.2 Command line Options

'`-s Stack`' The initial size of the stack can be specified. The unit is K-cells, with the default being 16 (K-cells); e.g. to double the size of the stack, specify '-s 32'.

‘`-h Heap`’ The initial size of the heap can be specified. The unit is K-cells, with the default being 48 (K-cells); e.g. to double the size of the heap, specify ‘-h 96’.

‘`-f File`’ Uses file `File` (instead of default `~`/icp.pl) as the startup file for Prolog component of ICP.

‘`-ff File`’ Uses file `File` (instead of default `~`/icp.par) as the initialisation file for Parlog component of ICP.

‘`-n`’ Skips Prolog startup file processing; i.e. does not read `~`/.icp.pl.

‘`-nn`’ Skips Parlog startup file processing; i.e. does not read `~`/.icp.par.

‘`-z String`’
`String` is executed by Prolog at startup time.

‘`-zz String`’
`String` is executed by Parlog at startup time.

‘`-F Charset`’
`Charset` may be one of ‘latin1’ (default), ‘greek’ or ‘mac’. This defines the character encoding which categorizes characters into upper case, lower case or graphic characters. This is used to determine variable names (beginning with upper case) and unquoted atoms (all lower case).

‘`-b File`’ Allows the selection of an alternative boot file. The boot file determines which system files are loaded at start up time and initialises the system. This option is rarely used.

1.3 Startup

When ICP is started it looks for the file ‘`$HOME/.icp.pl`’ which if found is consulted. This is commonly used to declare operators, load specific foreign code or prolog libraries. Lines of commands to be executed at startup time should be preceded by ‘:-’ or ‘?-’. ‘:-’ and ‘?-’ commands may be scattered throughout the file, but the behaviour is as if all the ‘:-’ commands (in the same order) are at the start of the file and all the ‘?-’ commands (in the same order) are at the end of the file. The clauses in the body of the file are asserted into the dynamic database. Commands that write output should specify the stream explicitly and not to rely on the default current output stream.

After the startup file has been processed, the commands passed from the command line through -z options are executed.

When the parlog sub-system is started up, it goes through a similar process using the startup file '`$HOME/.icp.par`' and the -zz options. Parlog may be started up in many ways - in the '`$HOME/.icp.pl`', as a -z command line option, or as a Prolog query.

1.4 Syntax

The syntax of ICP is compatible with Quintus Prolog[1] *Quintus Prolog Development Environment*, Quintus Computer Systems, Inc., Mountain View, California, USA.

1.4.1 Integers

An integer consists of a sequence of digits optionally preceded by a minus sign '`-`'. These are normally interpreted as base 10 integers. It is possible to enter integers in other bases (2 through 36). To enter an integer in a base other than 10, first enter the base in decimal, followed by an apostrophe ('`'`') and then the string of digits. If a base greater than 10 is used, the characters '`a`'-'`z`' or '`A`'-'`Z`' are used to stand for digits 10 through 35. For example:

```
8'21 = 17 decimal
16'FF = 255 decimal
36'Z = 35 decimal
```

As a special case, base 0 allows characters to represent their ASCII codes. Examples:

```
0'A = 65
0'[ = 91
```

1.4.2 Floating Point Numbers

A floating point number consists of a sequence of digits with an embedded decimal point ('`.`'), optionally preceded by a minus sign ('`-`') and optionally followed by an exponent consisting of an upper or lower case '`e`' and a signed base 10 integer. Note that there must be at least one digit before, and one digit after, the decimal point.

[1] Quintus is a trademark of Quintus Computer Systems Inc., USA

1.4.3 Atoms

An atom may take any of the following forms:

1. A sequence of alphanumeric characters (including underscore), starting with a lower case letter
2. A sequence of one or more characters from the set: `# $ & * + - . / : < = > ? \ ^ ‘ ~`
3. A sequence of characters delimited by single quotes (‘’). If the single quote character is included it must be written twice.
4. the special atoms `! ;` or `{}`

1.4.4 Variables

A variable is a sequence of alphanumeric characters (including underscore) starting with a capital letter or an underscore (‘_’). Variables which only occur once in the clause are called anonymous variables and may be written as an underscore on its own.

1.4.5 Tuples

A tuple is written as functor (which may be any term), followed by a left bracket ‘(’, followed by 0 or more comma separated terms, and ending with a right bracket ‘)’. Note that the functor is not limited to being an atom as in the case of Quintus Prolog. There must be no whitespace characters between the functor and the left bracket. The following are valid ICP tuples:

```
foo(X)
bar()
3(a,b,c)
Var(foo,bar)
person(john,33)(X,Y,Z)
```

Note that a tuple with 0 arity is not the same as the functor by itself i.e. `X() = X` fails.

Atomic functors may be declared as operators which would allow the tuple to be written without the brackets. Operators may be declared to be prefix, postfix or infix. Infix operators can be further defined as none, left or right associative.

1.4.6 Lists

A list is just a shorthand notation for '`./2`' tuples. A list is written as a left square-bracket '`[`' followed by 0 or more comma separated terms and ending with a right-square bracket '`]`'. The list may also be written in '`./2`' notation. e.g. `[1,2,3]` unifies with `.(1,.(2,.(3,[])))`. For convenience, a list of ASCII codes may be written as a string delimited by double-quotes ('`"`'). e.g. `"ABC" = [65,66,67]`. To include the double-quote character in the string, it must be written twice.

1.4.7 Character Escaping

In quoted atoms, double-quote delimited lists or base 0 notation, escape sequences can be used to denote characters. This is used to avoid ambiguity and to enter unusual characters. ICP has the following escape sequences:

`\a` alarm/bell (ASCII 7)

`\b` backspace (ASCII 8)

`\t` tab (ASCII 9)

`\n` newline (ASCII 10)

`\v` vertical tab (ASCII 11)

`\f` formfeed (ASCII 12)

`\r` carriage return (ASCII 13)

`\e` escape (ASCII 27)

`\s` space (ASCII 32)

`\d` delete (ASCII 127)

`\<octal string>`
: the character with ASCII code `<octal string>` base 8. e.g. `\007` is the bell character and `\040` is the space character (ASCII 32).

`\^<control char>`
: the character whose ASCII code is the `<control char>` mod 32. e.g. `\^P` is `CTL-P`.

`\<layout char>`
: no character, where `<layout char>` is a character with ASCII code =< 32 or >= 127. This is most useful when splitting atoms over two lines where the ignored newline character is preceded by a backslash.

`\c` no character, also all characters up to, but not including, the next non-layout character are ignored.

`\<other>` the character `<other>`, where `<other>` is any character not defined above.

1.5 Notation

Built-in predicates are introduced with a template such as **bag_of**(*?X*, *+Goal*, *-Set*) where *X*, *Goal* and *Set* are the arguments. The significance of the leading characters is as follows.

`+` The argument is a (non variable) input.

`-` The argument is an output. The output value is unified with the original value.

`?` The argument is neither of the above. This might be because the argument can be used both as an input and as an output argument, or because while the argument is an input, it can be an uninstantiated variable.

Often, a tuple or predicate is represented by the term *Name/Arity*, where *Name* is the function symbol and *Arity* is the number of arguments of the term.

```
Term            Representation
----            --------------
foo(X,Y)        foo/2
bar             bar/0
not Goal        not/1
1+2             +/2
```

2 Builtin Predicates

2.1 Control

`+`*C1*`, +`*C2* This is the conjunction operator. *C1* is evaluated and then *C2* is evaluated. The conjunction succeeds if and only if both calls succeed.

`+`*C1* `; +`*C2*

This is the disjunction operator. The call succeeds if either *C1* or *C2* succeeds. *C1* is evaluated first. Only when all possible evaluation paths for C1 have been explored will backtracking lead to an evaluation of *C2*.

A `!/0` evaluated in the disjunction not only prevents backtracking within the disjunction, it also prevents backtracking to find alternative solutions to calls that precede the disjunction in the clause or query in which the disjunction appears.

A `!/0` evaluated within the *C1* branch also prevents the use of *C2*.

`!` This is the backtracking control primitive. After it has been evaluated in a clause a backtrack to the `!/0` call will be interpreted as failure of the call C that invoked the clause. That is, it prevents the search for alternative solutions to any calls that precede the `!/0` in the body of the clause, and it will also prevent the use of other clauses to try to solve C.

Placed at the top level in a query conjunction it will prevent backtracking to find alternative solutions to calls that precede it in the query. See the description of the logical operators `;/2` and `->/2` for the effect of `!/0` inside these operators.

`call(+`*Goal*`)`

The call is equivalent to the call *Goal*, except where *Goal* is `!/0` or includes `!/0`. A `!/0` evaluated inside *Goal* only has a local effect. It only affects backtracking to find alternative solutions to calls that precede the `!/0` within *Goal*.

`not +`*Call* This is the negation as failure operator. The call succeeds if and only if *Call* fails.

`\+ +`*Call* Same as `not/1`.

`+`*Test* `-> +`*Call*

This is the conditional evaluation operator. The `->/2` call succeeds if and only if *Test* succeeds and *Call* succeeds. An `if` ... `then` ... `else` construction may be created by combining the conditional evaluation operator with the disjunction operator '`;/2`' i.e. *Test* `->` *Call* `;` *Else*.

As with disjunctions and conjunctions, a `!/0` evaluated in *Test* or *Call* not only has a local effect, but also the same effects as a `!/0` evaluated just before the `->/2` call.

`true` This call always succeeds.

`otherwise`
Same as `true/0`.

`fail` This call does not match anything and always fails.

`false` Same as `fail/0`.

`repeat` This call always succeeds and repeatedly succeeds each time the evaluation backtracks to the call. It is defined as:

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

`succeed(+`*Call*`)`
Call is evaluated and `succeed/1` succeeds, regardless of whether *Call* succeeded or failed. Its definition is:

```
succeed(Call) :-
        Call,
        !.
succeed(Call).
```

`one(+`*Goal*`)`
Goal must be a call term: `one/1` is used to find just one solution to *Goal*.

2.2 Loading, Consulting and Compiling

2.2.1 Introduction

In ICP there is a distinction between dynamic (consulted) and static (compiled) code. When a file is loaded into the runtime system, it is stored either in the system or user partition. Predicates in the system partition can not be modified as they are necessary for running the prolog system. Dynamic code is allways in the user partition.

2.2.1.1 The Search Path

Source files are assumed to have the suffix '`.pl`' while compiled files have the extension '`.icp`'. Many predicates dealing with files will automatically append these extensions to files if they are not present. Thus for example the goal

```
?- consult(foo).
```

will result in the file '`foo.pl`' being consulted.

The locations in the Unix file system which will be searched to locate the specified file are determined by the Unix environment variable '`ICP_PATH`'. This must be a colon separated list, for example

```
ICP_PATH=.:$HOME/icp/binary:$HOME/icp/source
```

ICP appends to the user defined path directories in the installation directory of ICP. If the variable '`ICP_PATH`' is not set, the search path is set as if the variable were set to the following value

```
ICP_PATH=.:$HOME/icp/binary:$HOME/icp/source:$ICP_INSTALLDIR/prolog/$ARCH
```

where '`$ICP_INSTALLDIR`' is the ICP instalation directory and '`$ARCH`' is the machine architecture as reported by the unix command '`/bin/arch`' (eg '`sun3`' or '`sun4`').

2.2.1.2 Dynamic Code

Dynamic code is code which can be modified dynamically by the user by `assert/1` and, `retract/1` (see Section 2.9 [ModifyingDatabase], page 36). In addition dynamic code is 'visible' in that it can be accessed by `clause/2` and listed by `listing/0` and `listing/1` (see Section 2.7 [ProgramState], page 29). Dynamic code can also be traced.

Dynamic code can be created either by consulting a file using `consult/1`, `reconsult/1` (see Section 2.2.3 [LoadingDynamic], page 10), or by the dynamically asserting clauses.

2.2.1.3 Static Code

Static code is generated by the compiler. In contrast to dynamic code it is 'hidden' from the user; the definition of a static clause can not be accessed directly and it is forbiden to dynamically modify it. Static code is optimized and thus executes much faster than dynamic code.

Compiled code is generated by the `compile/2` and `compile/3` calls (see Section 2.2.4 [LoadingStatic], page 11). The compiler saves the ICP object code in a file with the extension '`.icp`'. Once a file has been compiled it must be loaded into the ICP process.

2.2.2 Style Warnings

ICP has a style checker which can help in detecting errors if a particular style is used. Thus is as follows.

1. All clauses for a predicate should be contiguous — clauses for one procedure should not be interspaced with clauses for another.
2. Variables which appear only once in a clause should be written as '`_`' or a name beginning with an '`_`'.

If this style is not adhered to ICP can write warnings to `user_error`.

```
WARNING: Clauses for a/1 are not together in the source file
WARNING: Singleton variables, clause  2 of b/1:  [X]
```

The style checking can be controled by the predicates `style_check/1` and `no_style_check/1`.

`style_check(`*Type*`)`

`all` turns on/off all style checking

`discontiguous`
: turns on/off checking for discontiguous clauses

`single_var`
: turns on/off checking for single occurence of variables. Variables whose name begins with an underscore (_) are not flagged.

`no_style_check(`*Type*`)`

2.2.3 Loading Dynamic Code

`consult(+`*File*`)`

File is either the atom `user`, or the name of some source file. In the case of *File* not being the atom `user`, the file is read and all clauses defined in it are asserted into the internal database at the end. Thus if a predicate defined in the file already has a definition in the internal database, the effect is to expand the definition of this predicate. Consultation terminates when the end of the file is reached.

If a syntax error is found during consultation then the string '`** SYNTAX ERROR **`' is written out followed by all the text up to the offending token, followed by the string '`*HERE*`', followed by all those tokens which proceed the next period or the end of the file being consulted. A syntax-error message is also displayed. Consultation continues after this period.

On encountering a clause for a system predicate or for a currently defined static predicate, `consult/1` ends the current session, displays an error message, and then fails.

In the special case where *File* is the atom `user` `consult/1` reads from the terminal (standard input) rather than a file. Consultation terminates when a `end_of_file` token is read (caused by typing `CTL-D`). If a syntax error is found during consultation then the string '`** SYNTAX ERROR **`' is written out, together with the line of text in which the error occurred. A syntax-error message is also displayed.

The top-level goal

```
?- [File1, ..., FileN].
```

is syntactic sugar for the goals

```
?- consult(File1), ..., consult(FileN).
```

`reconsult(+`*File*`)`

Like `consult/1` except that all predicates previously (re)consulted for this file are abolished before asserting the new definitions.

2.2.4 Loading Static Code

`compile(+`*File*`)`

`compile/1` is used to generate static object code for the relations in *File*. *File* is either the atom `user`, or the name of some source file. The predicates defined in file *File* are compiled and written out to the corresponding object file. Completion of suffixes is performed, with the object file having the extension '`.icp`'.

`compiling user`

If *File* is `user` then the relations the user wishes to generate code for are to be entered directly at the terminal. Each relation is compiled to a separate segment which is automatically loaded. If a syntax error is found during

compilation then the string '** SYNTAX ERROR **' is written out, together with the line of text in which the error occurred. A syntax-error message is also displayed. Compilation ends when the user types a CTL-D.

compiling a '.pl' file

If *File* is the name of a '.pl' file then compile/1 obtains the relations the user wishes to generate code for by reading the file in question. All the relations are compiled into a single segment which, on successful completion of compilation is written out to the file '*File*'.icp''. If a syntax error is found during compilation then the string '** SYNTAX ERROR **' is written out, followed by all the text up to the offending token, followed by the string '*HERE*', followed by all those tokens which precede the next period or the end of the file being compiled. A syntax-error message is also displayed. compile/1 then continues reading from the first character following the period. Compilation ends when an end of file is reached. On encountering an invalid clause, compile/1 displays an error message and continues as if the offending clause had not been read.

compile(+*File*,+*Public*)

Same as compile/1 except that on loading the code resulting from compilation, only the successfully compiled relations of *File* whose predicates appear in the *Public* list, become accessible other than to predicates in the same segment. Predicates of arity zero may appear in *Public* as simple atoms.

compile(+*File*,+*Public*,+*Mode*)

Same as compile/2 except that the third argument allows for non-standard compilation modes to be adopted. If *Mode* is the atom single then the relations of *File* are compiled into a single segment, otherwise *Mode* must be the atom multiple and each relation of *File* is compiled into a separate segment.

Single mode generates faster and more compact code, but multiple mode is more flexible because individual predicates may be redefined. An analogy is that predicates compiled in multiple mode are like black boxes while those compiled in multiple mode have replaceable parts. The default is single mode for compiling files and multiple mode for compiling user.

load(+*File*)

If *File* is an atom, loads the object code contained in file '*File*'.icp''. load/1 notifies the user of the file being loaded and returns after typing the word 'done'. Files containing built-in predicates are not loaded (a warning is given). If the file to be loaded contains any predicate which is already defined, then the predicate is abolished before loading takes place. The '.icp' files are first looked for in the current directory, and then in the subdirectory "binary''. If the source file is newer than the compiled file, the source file is compiled before the load.

If *File* is a list, is equivalent to repeatedly calling `load/1` with the members of *Files*.

`load(+`*File*`, +`*Type*`)`
: As `load/1` but of type *Type*, where *Type* is one of the atoms `system` or `user` determining whether the predicates should be placed in a system or user partition.

`ensure_loaded(+`*File*`)`
: If the filename *File* is an atom, and the file has not been loaded yet, this behaves in the same way as `load/1`. If the file was previously loaded, then `ensure_loaded/1` succeeds trivially.

  If *File* is a list, this is equivalent to repeatedly calling `ensure_loaded/1` with the members of *Files*.

`loadicp(+`*File*`)`
: Loads a compiled file. This is a form of loading which disables automatic compilation and which does not handle lists of files. If *File* does not have the suffix '`.icp`' then *File*.icp is loaded.

`loadicp(+`*File*`, +`*Type*`)`
: As `loadicp/1` but of type *Type*, where *Type* is one of the atoms `system` or `user` determining whether the predicates should be placed in a system or user partition.

`rebuild(+`*File*`)`
: Is used to recompile all system files that need it.

`make(+`*File*`)`
: Is used to recompile a file if the source file is newer than the object file. All predicates defined in the file *File* are made public.

`make(+`*File*`, +`*Public*`)`
: Is used to recompile a file if the source file is newer than the object file. All predicates in the list *Public* are made public, unless *Public* is the empty list (`[]`) in which case all predicates in the file are made public.

`system(+`*OnOff*`)`
: Normally system predicates can not be re-loaded, an attempt to do so results in a message of the form

```
| ?- compile(user).
write(A):-foo(A).
cannot redefine system predicate(s): write/1.
```

  The call `?- system(on).` causes subsequent calls to `compile/2` to allow redefinition of system predicates. To restore the normal operation, the call `?- system(off).` should be used.

2.3 Input and Output Primitives

Input and output can be performed either on Prolog terms or on individual characters. Furthermore output is relative to *streams* which refer either to files or to the terminal. The system always has a *current input* and *current output* stream, which are by default the keyboard and screen respectively.

2.3.1 Input and Output of Terms

Most of these predicates use the current input or output streams but have a version where the stream may be specified. In this case the extra argument for the stream, *Stream*, is the first argument of the predicate. The stream versions of the predicates are implemented in terms of the current input/output versions and the predicates of section Section 2.3.3 [StreamHandling], page 21. Hence if multiple reads or writes are to be performed on a stream, it is more efficient to set the current input or output to the stream than to use the stream I/O primitives.

`read(-`*T*`)`
`read(+`*Stream*`,-`*T*`)`

> *T* is unified with the next term that can be read (wrt the current operator declarations) from the current input stream. The term must be followed by a period followed by a layout character. `read/1` does nothing until these characters, which are consumed but are not a part of the term, have been read. Any variable names in the read-in term are converted into variables.
>
> If a syntax error is found, an error message is written out to `user_error`. If the current input stream is `user_input` then only the line of text in which the error occurred is displayed. If instead the current input stream is associated with a file, then all the text up to the offending token is displayed, followed by the string '`*HERE*`', followed by all those tokens which precede the next period or the end of file. `read/1` then tries again as if it had started reading from the first character following the period. Eventually, either some term is successfully read, or an end of file is reached, whence *T* is unified with the `end_of_file` atom. If the input stream is a terminal, `read/1` will suspend if there are no characters to be read.

`write(?`*X*`)`
`write(+`*Stream*`,?`*X*`)`

> Term *X* is written to the current output stream (wrt the current operator declarations). Terms of the form `'$VAR'(N)` are treated in a special way: On N being 0, 1, ..., 27, 28, ... write outputs '`A`', '`B`', ..., '`A1`', '`B1`', etc.

`writeq(?X)`
`writeq(+Stream,+X)`
`writeq(+Stream, ?X, +VarNames)`

Same as `write/1` except that it quotes those atoms that would need to be quoted on input.

In the 3 argument version, atoms in the list *VarNames* are not quoted. This gives a way of printing the original variable names provided the term was read using `gread/3`.

`display(?X)`
`display(+Stream,+X)`

Term *X* is written to the current output stream ignoring operators. Variables are displayed as underscore names. This is particularly useful for seeing the structure of terms containing multiple operators.

`write_canonical(?X)`
`write_canonical(+Stream,+X)`

Term *X* is written to the current output stream ignoring operators. Variables are displayed as underscore names. Atoms that would need to be quoted on input are quoted. This is useful for writing out terms that will be read back using `read/1-2`.

`portray(?Term)`

This is a user defined predicate which can be used to override the default system ways of printing terms.

`print(?Term)`
`print(+Stream, ?Term)`

This primitive allows user control over the output of terms. The print primitive looks for the program `portray/1`. If there is no definition for portray, the effect of print is exactly the same as for `writeq/1`.

If the portray program is defined, then it is called to output the *Term*. If `portray/1` succeeds, then `print/1` succeeds and exits.

If the call to `portray/1` fails, and *Term* is atomic, then *Term* will be output using `display/1`.

If the call to `portray/1` fails, and *Term* is compound, then the principal functor of *Term* will be output using `display/1`, and `portray/1` will be called recursively on the arguments of *Term*.

If the call to `portray/1` fails, and *Term* is a list, then the list will be output using bracket notation with `print/1` being called for each element.

`format(Format, Arguments)`
`format(Stream, Format, Arguments)`

Provides a means of writing formatted data. The following description is extracted from the Sicstus Prolog on-line manual.

Print *Arguments* onto *Stream* according to format *Format*. *Format* is a list of formatting characters. If *Format* is an atom then `name/2` will be used to translate it into a list of characters. Thus

```
| ?- format("Hello world!", []).
```

has the same effect as

```
| ?- format('Hello world!', []).
```

`format/3` is a Prolog interface to the C `stdio` function `printf()`. It is due to Quintus Prolog.

Arguments is a list of items to be printed. If there is only one item it may be supplied as an atom. If there are no items then an empty list should be supplied.

The default action on a format character is to print it. The character `~` introduces a control sequence. To print a `~` repeat it:

```
| ?- format("Hello ~~world!", []).
```

will result in

```
Hello ~world!
```

A format may be spread over several lines. The control sequence `\c` followed by a `LFD` will translate to the empty string:

```
| ?- format("Hello \c
world!", []).
```

will result in

```
Hello world!
```

The general format of a control sequence is '~NC'. The character C determines the type of the control sequence. N is an optional numeric argument. An alternative form of N is '`*`'. '`*`' implies that the next argument in *Arguments* should be used as a numeric argument in the control sequence. Example:

```
| ?- format("Hello~4cworld!", [0'x]).
```

and

```
| ?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available.

'`~a`' The argument is an atom. The atom is printed without quoting.

'~N`c`' (Print character.) The argument is a number that will be interpreted as an ASCII code. N defaults to one and is interpreted as the number of times to print the character.

'~N`e`'
'~N`E`'
'~N`f`'

'~*N*g'

'~*N*G' (Print float). The argument is a float. The float and *N* will be passed to the C `printf()` function as

```
printf("%.Ne", Arg)
printf("%.NE", Arg)
printf("%.Nf", Arg)
printf("%.Ng", Arg)
printf("%.NG", Arg)
```

If *N* is not supplied the action defaults to

```
printf("%e", Arg)
printf("%E", Arg)
printf("%f", Arg)
printf("%g", Arg)
printf("%G", Arg)
```

'~*N*d' (Print decimal.) The argument is an integer. *N* is interpreted as the number of digits after the decimal point. If *N* is 0 or missing, no decimal point will be printed. Example:

```
| ?- format("Hello ~1d world!", [42]).
| ?- format("Hello ~d world!", [42]).
```

will print as

```
Hello 4.2 world!
Hello 42 world!
```

respectively.

'~*N*D' (Print decimal.) The argument is an integer. Identical to '~*N*d' except that ',' will separate groups of three digits to the left of the decimal point. Example:

```
| ?- format("Hello ~1D world!", [12345]).
```

will print as

```
Hello 1,234.5 world!
```

'~*N*r' (Print radix.) The argument is an integer. *N* is interpreted as a radix. *N* should be >= 2 and <= 36. If *N* is missing the radix defaults to 8. The letters 'a-z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~2r world!", [15]).
| ?- format("Hello ~16r world!", [15]).
```

will print as

```
Hello 1111 world!
Hello f world!
```

respectively.

'~*N*R' (Print radix.) The argument is an integer. Identical to '~Nr' except that the letters 'A-Z' will denote digits larger than 9. Example:

```
| ?- format("Hello ~16R world!", [15]).
```

will print as

```
Hello F world!
```

'~*N*s' (Print string.) The argument is a list of ASCII codes. Exactly *N* characters will be printed. *N* defaults to the length of the string. Example:

```
| ?- format("Hello ~4s ~4s!", ["new","world"]).
| ?- format("Hello ~s world!", ["new"]).
```

will print as

```
Hello new  worl!
Hello new world!
```

respectively.

'~i' (Ignore argument.) The argument may be of any type. The argument will be ignored. Example:

```
| ?- format("Hello ~i~s world!", ["old","new"]).
```

will print as

```
Hello new world!
```

'~k' (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/2`. Example:

```
| ?- format("Hello ~k world!", [[a,b,c]]).
```

will print as

```
Hello .(a,.(b,.(c,[]))) world!
```

'~p' (print.) The argument may be of any type. The argument will be passed to `print/2`. Example:

```
| ?- assert((portray([X|Y]) :- print(cons(X,Y)))).
| ?- format("Hello ~p world!", [[a,b,c]]).
```

will print as

```
Hello cons(a,cons(b,cons(c,[]))) world!
```

'~q' (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/2`. Example:

```
| ?- format("Hello ~q world!", [['A','B']]).
```

will print as

```
Hello ['A','B'] world!
```

'~w' (write.) The argument may be of any type. The argument will be passed to `write/2`. Example:

```
| ?- format("Hello ~w world!", [['A','B']]).
```

will print as

```
Hello [A,B] world!
```

'~*N*n' (Print newline.) Print *N* newlines. *N* defaults to 1. Example:

```
| ?- format("Hello ~n world!", []).
```

will print as

```
Hello
 world!
```

`gread(-`*T*`)`
`gread(+`*Stream*`,-`*T*`)`
`gread(+`*Stream*`,-`*T*`,-`*V*`)`

Same as `read/1` except that the variable names which occur in the read-in term appear in *T* as quoted atoms, rather than being converted into variables.

In the 3 argument version *V* is instantiated to the list of atoms that are the variable names of the read-in term.

```
| ?- gread(user_input, T, V).
foo(X, a(Y, X)).

T = foo('X',a('Y','X'))
V = ['X','Y']
```

`writeseq(+`*X*`)`
`writeseq(+`*Stream*`,+`*X*`)`

Uses `write/1` to write the elements of the list *X* in sequence and separated by single spaces.

`writeqseq(+`*X*`)`
`writeqseq(+`*Stream*`,+`*X*`)`

Uses `writeq/1` to write the elements of the list *X* in sequence and separated by single spaces.

`writenl(?`*X*`)`
`writenl(+`*Stream*`,+`*X*`)`

Starts a new line on the current output stream after having used `write/1` to write *X*.

`writeqnl(?`*X*`)`
`writeqnl(+`*Stream*`,+`*X*`)`

Starts a new line on the current output stream after having used `writeq/1` to write *X*.

`writeseqnl(+`*X*`)`
`writeseqnl(+`*Stream*`,+`*X*`)`

Starts a new line on the current output stream after having used `write/1` to write the elements of *X* in sequence and separated by single spaces.

```
| ?- writeseqnl([a1,'A2',4]).
a1 A2 4

yes
```

`writeqseqnl(+`*X*`)`
`writeqseqnl(+`*Stream*`,+`*X*`)`
: Starts a new line on the current output stream after having used `writeq/1` to write the elements of *X* in sequence and separated by single spaces.

`write_term(?`*Term*`)`
`write_term(+`*Stream*`, ?`*Term*`)`
: Same as `write/1` except that the term will be written in an encoded form which can only be decoded by `read_term/1`. This is more efficient than `write/1` because the structure of the term is preserved in the encoding.

`read_term(-`*Term*`)`
`read_term(+`*Stream*`, -`*Term*`)`
: Reads an encoded term from the current input stream. The encoded version of a term is read by `read_term/1` at a greater speed than `read/1` can read the original term. `write_term/1` and `read_term/1` can be used to write to files large terms that require frequent reading.

`portray_clause(?`*Clause*`)`
: Displays a clause in the same way as `listing/1`. In fact `listing/1` and `listing/2` use `portray/1`.

2.3.2 Input and Output of Characters

Most of these predicates use the current input or output streams but have a version where the stream may be specified. In this case the extra argument for the stream, *Stream*, is the first argument of the predicate. The stream versions of the predicates are implemented in terms of the current input/output versions and the predicates of section Section 2.3.3 [StreamHandling], page 21. Hence if multiple reads or writes are to be performed on a stream, it is more efficient to set the current input or output to the stream than to use the stream io primitives.

`get0(-`*C*`)`
`get0(+`*Stream*`, -`*C*`)`
: Reads the next character (ASCII code) from the current input stream and instantiates *C* to it. The predicate returns 26 (`CTL-Z`) on `end_of_file`.

`get(-`*C*`)`

`get(+`*Stream*`, -`*C*`)`
: Reads the next character (ASCII code) from the current input stream and instantiates *C* to it, ignoring non printable characters (ascii code < 33 or > 126). The predicate returns 26 on `end_of_file`.

`skip(+`*N*`)`
`skip(+`*Stream*`, +`*N*`)`
: Skips over characters from the current input stream to the first occurrence of the character with ASCII code *N*.

`put(+`*N*`)`
`put(+`*Stream*`, +`*N*`)`
: Writes *N* to the current output stream. *N* should be either a valid ASCII code or an integer expression.

`nl`
`nl(+`*Stream*`)`
: Starts a new line on the current output stream.

`tab(+`*N*`)`
`tab(+`*Stream*`, +`*N*`)`
: N (>=0) spaces are written to the current output stream.

`tty_get0(-`*C*`)`
: Reads the next character (ASCII code) from the current terminal input stream and instantiates *C* to it. This is used when an input character must come from the keyboard, not necessarily the current stream.

`unget(+`*N*`)`
: Puts ASCII code *N* back onto the current input stream.

2.3.3 File and Stream Handling

Input and output streams are represented as unique prolog terms having the following form where `N` is an integer.

`stream(N)`
: refers to the terminal or files

`memory(N)`
: refers to ram files, which can be converted to and from prolog atoms.

There are also three streams which are opened automatically by the system

`user_input`
: The standard input channel ('C' `stdin`)

`user_output`
: The standard output channel ('C' `stdout`)

`user_error`
: The standard error channel ('C' `stderr`)

`open(+`*F*`,+`*M*`,-`*Stream*`)`
: Opens file *F* in mode *M* on file-stream *Stream*. *M* must be one of `read`, `write` and `append`.

`close(+`*Stream*`)`
: Closes the stream corresponding to *Stream*.

`set_input(+`*Stream*`)`
: Makes *Stream* the current input stream.

`set_output(+`*Stream*`)`
: Makes *Stream* the current output stream.

`current_input(-`*Stream*`)`
: Instantiates *Stream* with the current input stream.

`current_output(-`*Stream*`)`
: Instantiates *Stream* with the current output stream.

`open_ram(+`*C*`,+`*M*`,-`*Stream*`)`
: Opens ram-file *C* in mode *M* on memory-stream *Stream*. *M* must be one of `read`, `write` and `append`. When opening a ram-file in `read` or `append` mode *C* must be an atom representing the file's contents. This atom is usually the result of a `ram_const/2` call. When opening the file in `write` mode, argument *C* is ignored and the file is created empty.

`open_ram(+`*C*`, +`*M*`, -`*Stream*`, +`*Size*`)`
: As `open_ram/3` but allows the size of the ram file to be specified.

`ram_const(+`*F*`, -`*C*`)`
: Converts the contents of the ram-file `+`*F* into an atom `-`*C*. Typically, `-`*C* is subsequently used as the first argument of an `open_ram/3` call to access the contents of the ram-file.

`ram_pipe(-`*Out*`,-`*In*`)`
: Creates a pseudo-pipe by allocating a ram file with two references to it. The first reference *Out* is opened in write mode, and the second reference *In* is opened in read mode. This allows one thread to pass terms to another. It is the responsiblity of the programmer to make sure that there is data available before it is read. For pipes with

blocking read and write, see `pipe/2` in chapter 14. The advantage of ram pipes is that they can be made the subject of current input and current output.

`tty(+`*Name*`,-`*Read*`,-`*Write*`)`
: Creates a new window with title *Name* and two streams — *Read* and *Write*. Input typed in the new window can be read from stream *Read*. Output written to stream *Write* will appear in the window.

`flush_output(+`*Stream*`)`
: *Stream* must be a stream open for output. *Output* to a stream is not necessarily sent immediately; it is buffered. This predicate flushes the output buffer for the specified stream and thus ensures that everything that has been written to the stream is actually sent at that point.

`flush`
: Same as `flush_output/1` except that the flushed stream is the current output stream.

`file_exists(+`*F*`)`
: Succeeds if file *F* exists.

2.4 Arithmetic

-*X* `is +`*Y*
: *X* is the result of evaluating arithmetic expression *Y*.

An arithmetic expression is either an number, or a variable which is bound to a number, or a singleton list whose head is an integer (in this case the value of the expression is the integer itself), or a single character in double quotes (in this case the value of the expression is the ASCII code of the character), or a compound expression. A compound expression is a tuple whose functor is an arithmetic operator and whose arguments are arithmetic expressions. A description of the allowed compound expressions, together with the result of their evaluation follows:

`-`*E*	negative of *E*
E1 `+` *E2*	sum of *E1* and *E2*
E1 `++` *E2*	integer sum of integers *E1* and *E2*
E1 `-` *E2*	difference of *E1* and *E2*
E1 `--` *E2*	integer difference of integers *E1* and *E2*
E1 `*` *E2*	product of *E1* and *E2*
E1 `**` *E2*	integer product of integers *E1* and *E2*
E1 `/` *E2*	quotient of *E1* and *E2*
E1 `//` *E2*	integer part of quotient of integers *E1* and *E2*

E1 `mod` *E2*
: remainder of the integer division of *E1* and *E2*

E1 `^` *E2*
: *E1* raised to the power *E2*

E1 `<<` *E2*
: integer *E1* shifted left by *E2* bits

E1 `>>` *E2*
: integer *E1* shifted right by *E2* bits

`\`*E*
: bitwise logical negation of integer *E*

E1 `/\` *E2*
: bitwise logical And of integers *E1* and *E2*

E1 `\/` *E2*
: bitwise logical Or of integers *E1* and *E2*

`int(`*E*`)`
: integer truncation of *E* towards 0

`abs(`*E*`)`
: absolute value of *E*

`sign(`*E*`)`
: sign of *E*: -1 if negative, 1 if positive and 0 if 0

`ceil(`*E*`)`
: round up to next higher integer

`floor(`*E*`)`
: round down to next lower integer

`sin(`*E*`)`
: sine of an angle *E* in radians

`cos(`*E*`)`
: cosine of an angle *E* in radians

`tan(`*E*`)`
: tangent of an angle *E* in radians

`sin(`*E*`)`
: arcsine of *E* (in radians) in the range -pi/2 to pi/2

`acos(`*E*`)`
: arcosine of *E* (in radians) in the range 0 to pi

`atan(`*E*`)`
: arctangent of *E* (in radians) in the range -pi/2 to pi/2

`atan2(`*E1*`, `*E2*`)`
: arctangent of *E1*/*E2* (in radians) in the range -pi to pi

`sinh(`*E*`)`
: hyperbolic sine of an angle *E* in radians

`cosh(`*E*`)`
: hyperbolic cosine of an angle *E* in radians

`tanh(`*E*`)`
: hyperbolic tangent of an angle *E* in radians

`exp(`*E*`)`
: natural anti-logarithm of *E*

`log(`*E*`)`
: natural logarithm of *E*

`log10(`*E*`)`
: logarithm (base 10) of *E*

`sqrt(`*E*`)`
: square root of *E*

`pow(`*E1*`, `*E2*`)`
: *E1* raised to the power *E2*

`pi` value of pi

`rand` random integer in the range 0 to 32767

`deg2rad(`*E*`)`
number of radians in *E* degrees

`rad2deg(`*E*`)`
number of degrees in *E* radians

`integer(`*E*`)`
the integer component (integer closest to 0) of *E*'s value.

`float(`*E*`)` *E*'s value.

2.4.1 Arithmetic Comparison

`+`*X* `=:=` `+`*Y*
X is equal to *Y*.

`+`*X* `=\=` `+`*Y*
X is not equal to *Y*.

`+`*X* `<` `+`*Y* *X* is less than *Y*.

`+`*X* `>` `+`*Y* *X* is greater than *Y*.

`+`*X* `=<` `+`*Y* *X* is less than or equal to *Y*.

`+`*X* `>=` `+`*Y* *X* is greater than or equal to *Y*.

2.5 Type Primitives

`tag(+`*Term*`,-`*Tag*`)`
Tag is bound to an integer in the range 0-6 which uniquely identifies *Term*'s type. The tag values are as follows:

Tag`=0` variable

Tag`=1` integer

Tag`=2` floating point

Tag`=3` atom

Tag`=4` nil (empty list)

Tag=5 list

Tag=6 tuple

`var(`*?X*`)` Succeeds if *X* is an uninstantiated variable.

`nonvar(`*?X*`)`
Succeeds if *X* is instantiated.

`atom(`*?X*`)` Succeeds if *X* is an atom.

`integer(`*?X*`)`
Succeeds if *X* is an integer.

`float(`*?X*`)`
Succeeds if *X* is a floating point number.

`number(`*?X*`)`
Succeeds if *X* is a number (integer or floating point).

`atomic(`*?X*`)`
Succeeds if *X* is an atom or a number.

`list(`*?X*`)` Succeeds if *X* is a list (empty or non-empty).

`compound(`*?X*`)`
The call succeeds if and only if *X* is bound to a tuple or a non-empty list.

`tuple(`*?X*`)`
Succeeds if *X* is a tuple.

`tuple(`*?X*`,-`*N*`)`
Succeeds if *X* is a functor of arity *N*-1 (*N* >= 1).

`length(`*?Term*`,`*?N*`)`
If *Term* is a list with a non variable length, *N* will be bound to its length.

If *N* is an integer then *Term* will be instantiated to a list with *N* elements.

If *Term* is a tuple, *N* will be unified with the tuple's arity plus one.

?T1 `=` *?T2*
Defined as if by the clause `"Z=Z."`; that is, *T1* and *T2* are unified.

?T1 `\=` *?T2*
Succeeds if an only if the call `T1=T2` fails; that is, if *T1* and *T2* are not unifiable.

`functor(`*?Term*`,`*?Name*`,`*?Arity*`)`
Term is a term with functor *Name* and arity *Arity*. Atomic terms are taken to have arity 0 and functor identical to the term. If *Term* is a variable the arity must be an integer >= 0 and *Name* a valid ICP tuple functor.

```
| ?- functor(foo(a1, a2, a3), Name, Arity).

Name = foo
Arity = 3

| ?- functor(Term, foo, 3).

Term = foo(_132,_133,_134)

| ?- functor(foo, Name, Arity).

Name = foo
Arity = 0

| ?- functor(T, f(Z), 2).

Unbound variable : Z
T = f(Z)(_113,_114)
```

`arg(+`*N*`,+`*Tup*`,-`*Arg*`)`

If *N* is a positive integer less than or equal to the arity of tuple *Tup* then *Arg* is unified with *Tup*'s *N*th argument.

```
| ?- arg(2, foo(a1, a2, a3), Arg).

Arg = a2
```

`nth0(`*?N*`,+`*List*`,`*?El*`)`

If *N* is a positive integer, this finds the nth element *?El* in a list *+List*, counting the first element as 0. If *N* is a variable, then the position of the element *?El* is calculated.

`nth1(`*?N*`,+`*List*`,`*?El*`)`

If *N* is a positive integer, this finds the nth element *?El* in a list *+List*, counting the first element as 1. If *N* is a variable, then the position of the element *?El* is calculated.

If *N* is a positive integer less than or equal to the arity of tuple *Tup* then *Arg* is unified with *Tup*'s *N*th argument.

?X `=..` *?Y*

Y is a list. Its head is the functor of *X* and its (possibly empty) tail is a list of *X*'s arguments.

2.6 Term Comparison

2.6.1 The Standard Order on Terms

If two terms are of a different type, then the ordering is as follows:
variable < number < symbol < nil < list < tuple

If two terms have the same type then:

- variables are compared by their machine address
- numbers are compared using '</2'.
- symbols are compared lexicographically, using the ASCII ordering.
- lists are compared by their first element. If equal, the tails are compared.
- tuples are compared first by arity, then by each of the arguments.

2.6.2 Term Comparison Predicates

`compare(`*?Op*`,`*?T1*`,`*?T2*`)`

The result of comparing terms *T1* and *T2* is *Op*, where the possible values of *Op* are:

'=' If *T1* is identical to *T2*,

'<' If *T1* is before *T2* in the Standard Order

'>' If *T1* is after *T2* in the Standard Order

?T1 `==` *?T2*

Succeeds if the terms currently instantiating *T1* and *T2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical).

?T1 `\==` *?T2*

Succeeds if the terms currently instantiating *T1* and *T2* are not literally identical.

?T1 `@>` *?T2*

Succeeds if term *T1* is after term *T2* in the Standard Order.

?T1 `@<` *?T2*

Succeeds if term *T1* is before term *T2* in the Standard Order.

?T1 `@>=` *?T2*

Succeeds if term *T1* is not before term *T2* in the Standard Order.

?T1 `@=<` *?T2*

Succeeds if term *T1* is not after term *T2* in the Standard Order.

2.7 Examining the Program State

`defined(+`*PredArity*`)`

Succeeds if *PredArity* is currently defined.

```
| ?- defined(defined/1).

yes
```

`predicate(`*?Pred*`,`*?Term*`)`

Succeeds if it can unify *Pred* with the name of a currently defined predicate and *Term* with the most general tuple with the same arity (i.e. a tuple with distinct variables for all of its arguments) and having *Pred* as its functor and the predicate's arity as its arity.

```
| ?- predicate(Pred, Term).

Pred = mod%f
Term = mod%f(_149,_150,_151)

| ?- predicate(predicate, Term).

Term = predicate(_143,_144)
```

`current_predicate(`*?Pred*`,`*?Term*`)`

Succeeds if it can unify *Pred* with the name of a user-defined predicate and *Term* with the most general tuple (i.e. a tuple with distinct variables for all of its arguments) having *Pred* as its functor and the predicate's arity as its arity.

```
| ?- current_predicate(Pred, Term).

Pred = system_file
Term = system_file(_153)

| ?- current_predicate(foo, Term).

no
| ?- assert(foo(1,2)), current_predicate(foo, Term).

Term = foo(_466,_467)
```

`system_predicate(`*?Pred*`,`*?Term*`)`

Attempts to unify *Pred* with the name of a system-predicate and *Term* with the most general tuple with the same arity (i.e. a tuple with distinct variables for all of its arguments) having *Pred* as its functor.

```
| ?- system_predicate(Pred, Term).

Pred = mod%f
Term = mod%f(_153,_154,_155)

| ?- system_predicate(listing, Term).

Term = listing ;

Term = listing(_145) ;

no
```

`predicate_property(`*?Term*`, `*?Property*`)`

Finds the property associated with predicate *Term*. A predicate can have more than one property. The possible properties are

`static` The predicate is static (i.e. compiled).

`dynamic` The predicate is dynamic (i.e. consulted or asserted).

`system` The predicate is a system predicate.

`user` The predicate is a user defined predicate.

A predicate can have more than one property

```
| ?- predicate_property(predicate_property(_,_), Property).

Property = static ;

Property = system ;

no
```

`clause(`*?Head*`,`*?Body*`)`

Searches the database for a clause whose head matches *Head* and whose body matches *Body*. This predicate is non-determinate and can be used to backtrack through all the clauses matching *Head* and *Body*. It fails when there are no (further) matching clauses in the database.

For the purpose of this matching, unit clauses (clauses with no body) are treated as if they had a body consisting of the single goal `true/0`.

`listing` Lists to the current output channel all the relations (if any) which are in the data base.

`listing(+`*Pred*`)`

All forms of `listing/1` can only be used for dynamic predicates.

If *Pred* is an atom, lists to the current output channel all the relations (if any) which are in the data base and have *Pred* as name.

If *Pred* is a term of the form *Name/Arity* then the predicates of the specified name and arity are listed.

If *Pred* is a term of the form *Name/Arity1-Arity2*, then listing lists the relations with name Name and arities in the specified range to the current output channel.

Pred may also be a list of terms of the above form. Thus the goal

```
?- listing([foo, bar/2]).
```

is equivalent to the goal

```
?- listing(foo), listing(bar/2).
```

`listfile(+`*File*`)`

Lists to the current output channel all predicates in file *File*. If *File* does not end in '`.pl`' this extension is appended to the file name.

2.8 Execution and Error Handling

Error handling in ICP is based on the idea of catch-and-throw. The query

```
catch(Goal)
```

behaves in exactly the same way as

```
call(Goal)
```

if *Goal* succeeds or fails. However, if an error occurs during the execution of *Goal*, then the behaviour is as if `system_error/2` was called, with the first argument being *Goal* and the second argument bound to the error number. The error numbers are listed in the appendix.

Note that `system_error/2` is called with the argument *Goal* even though the error might have occurred in a subgoal of *Goal*. e.g. in the program,

```
p :- q, r.
q :- s, t.
```

and the call

```
catch(p)
```

the system error handler will report the error occurring in p even though the error might have occurred in the subgoal t. This has the advantage that the low-level definition of p can be hidden. If the user really wanted to report t as the culprit goal, this can be specified by changing the definition of q to be

```
q :- s, catch(t).
```

This introduces another nesting level of 'catch'. Errors are always reported at the innermost catch level.

Errors are normally generated from the execution of built-in primitives. Typically, a primitive is called with the wrong arguments. However, it is possible to manually force errors by using `throw/1`, where the argument specifies the error number. This is also the mechanism for generating any user-defined errors.

There are cases when we would like to call the system error handler with a different goal from the one which 'caught' the error. We can do this using `catch/2`.

If the user wishes to handle specific errors but default to the system error handler for all other cases, then the `user_error/2` hook should be used.

There is yet another form `catch/3` which offers the most control in error-handling. In this form, any goal may be called as the error handler to be used.

These different options are described in more detail below.

`catch(+`*Goal*`)`
: This defines a catch point. *Goal* is executed. If *Goal* succeeds without any errors, the call to `catch/1` succeeds. If *Goal* fails without any errors, the call to `catch/1` fails. Otherwise, if an error occurs during the execution of *Goal*, the system error handler of ICP is invoked with *Goal* and the error number.

`throw(+`*Error*`)`
: An error with error code *Error* is forced. The error is handled by the error handler associated with the most recent call to catch.

`catch(+`*Goal*`,+`*BadGoal*`)`

Same as `catch/1` except that the ICP error handler is invoked with *BadGoal*, rather than *Goal*, being the culprit goal.

`catch(+`*Goal*`,+`*ErrorHandler*`,-`*Error*`)`

Goal is executed. If *Goal* succeeds without any errors, the call to `catch/3` succeeds. If *Goal* fails without any errors, the call to `catch/3` fails. Otherwise, if an error occurs during the execution of *Goal*, *Error* is bound to the appropriate error code and then *ErrorHandler* is executed. This predicate is a hook for enabling user-defined error handling. This is very powerful because the user-defined error handler may contain any number of arguments. For simple cases where the error goal and the error number are sufficient, the `user_error/2` hook described next is better.

`user_error(+`*Goal*`, +`*Errno*`)`

Error handlers may be defined by `user_error/2`, where the first argument is the goal at the catch point (not necessarily the goal that caused the error) and the second argument is the error number.

Here is a sample error handler. Note the use of cut for each clause, and the final catch-all clause that passes the error to the error handler at the next level up (typically the system error handler).

```
user_error(Goal, 403) :- !, /* predicate not defined error */
writeseqnl([Goal, 'not defined yet']),
fail.
user_error(Goal, 600) :- !, /* divide by 0 error */
writenl('divide by zero is not possible'),
fail.
user_error(Goal, Error) :- !, /* throw to the next level up */
throw(Error).
```

For convenience, access to the system error handlers is provided via `system_error/2` and `system_fail/2`. The difference between the two is that `system_error/2` will cause a return to the top level ICP prompt, whereas `system_fail/2` will simply fail and cause a backtrack. Typically, these will be used in user-defined error handlers for the 'uninteresting' cases.

`system_error(+`*Goal*`, +`*Error*`)`

Display an error message for the error number *Error* and return to the top level prompt.

`system_fail(+`*Goal*`, +`*Error*`)`

Display an error message for the error number *Error* and then fail.

Undefined predicates are handled slightly differently. Initially they are not errors as such, but you may define them to be errors. So what is so special about undefined predicates ? Recall that a call

```
catch(p)
```

creates a catch point. Any errors that occur while executing p will cause a throw to the most recent catch point where it is reported. The part of the search tree between the catch and the throw (where the error occurred) is discarded. This is important because it means that whatever action we can take to recover from the error, it can only proceed from the 'catch' and not from the point where the error occurred (which may be deep down the search tree). This is fine for most errors where we want to continue from a known point, however for undefined predicates it is not ideal. Consider the following program :

```
p :- q.
q :- r.
r :- s.
```

and the call

```
| ?- catch(p).
```

(Actually the 'catch' is redundant here because all top-level queries are automatically caught) The system will report the error as

```
ERROR 403 : undefined predicate : p/0
```

This is confusing because it is 's' that is undefined, not 'p'. However, this is what happens because the only catch point created was at 'p'. Moreover, even if we defined our own error handler to try to recover, we cannot continue from 's', but instead we are forced to restart from 'p' again. The solution is 'obvious' - catch everything ! This solution is very expensive, because it means we use meta-call for everything. There must be a better way ...

The solution we adopted in ICP is to consider undefined predicates as interrupts rather than errors. With interrupts, we always know the exact point of interruption and we have the option to continue execution. This is more general than the above, because we can get the same behaviour by making the default action of the interrupt handler to throw error 403 (undefined predicate error). Of course, we can also do other things like load/consult files which may contain definitions for the culprit goal, or even arbitrarily succeed, fail or bind the culprit goal interactively to continue the execution. The interrupt handler just described is exactly the default action in ICP.

This default action may be changed by setting the 'undefined' property of the thread. Each thread has a thread identifier which may be retrieved by the call

```
thread(-Id)
```

By default, this property is not set and the interactive handler is invoked whenever an undefined predicate is encountered. If the property is set as follows :

```
set_prop(Id, undefined, fail)
```

then all undefined predicates are assumed to fail silently.

```
set_prop(Id, undefined, warning)
```

causes all undefined predicates to fail, but a warning message is displayed.

```
set_prop(Id, undefined, error)
```

treats all undefined predicates as errors, which means that an error is thrown to the most recent catch point. Note that in this case the exact location of the undefined predicate will be lost as explained previously.

Even if we installed our own `user_error/2`, undefined predicates will not call the new error handler because undefined predicates are not (by default) errors. To invoke the user-defined error handler, you will need to specify undefined predicates as being errors. We do this like so :

```
thread(Id), set_prop(Id, undefined, error)
```

as discussed previously.

Now this invokes our error handler as expected. However, we still have the problem of the catch point not being where the undefined predicate occurred. (This may not be a problem if the programmer has taken care to put catch calls around all potentially undefined predicates. In this case, no more action is required.) What we need is not just a replacement for the error handler, we need a replacement for the 'undefined-predicate-interrupt-handler' too !

This interrupt handler is specified using the `user_undefined/1` hook.

`user_undefined(+`*Goal*`)`

This is a hook to allow users to trap undefined predicate errors. The argument is the undefined goal. The user may specify actions to take and perhaps retry the goal. Note that this is independent of any error handler since undefined predicates are not errors unless explicitly thrown as in the example below.

```
user_undefined(Goal) :-
functor(Goal, Pred, Arity),
writeseqnl(['missing predicate :', Pred/Arity]),
throw(403).  /* error code for undefined predicate */
```

`system_undefined(+`*Goal*`)`

For convenience, access to the system undefined predicate interrupt handler is provided. This will typically be called from a `user_undefined/1` relation.

`halt` Exits ICP.

`abort` Aborts execution and, after warning the user, returns control to the top-level query handler.

`op(+`*Priority*`,+`*Type*`,+`*Opname*`)`

Declares an operator with name *Opname*. *Priority* should be an integer between 1 and 1200, which will determine the operator precedence when the operator's arguments are themselves operator expressions. The lower the priority number, the tighter binding the operator. The type argument is one of

`fx` non associative prefix operator. The subexpression forming the argument of the operator must have lower precedence than the operator itself.

`fy` associative prefix operator. The subexpression forming the argument can have the same precedence as the operator.

`xf` non associative postfix operator.

`yf` associative postfix operator.

`xfx` non associative infix operator.

`xfy` right-associative infix operator.

`yfx` left-associative infix operator.

Note that an operator may be declared to be prefix, postfix and infix all at the same time, though this is confusing and not recommended.

The precedence and type of an operator may be changed by re-declaring it. To remove an operator, re-declare it with precedence 0 for the relevant type.

When several operators have the same declaration, the *Opname* argument can be a list of operator names.

`current_op(`*?Priority*`,`*?Type*`,`*?Opname*`)`

If *Opname* is an atom, the call can be used to find the priority and type of a given operator. If *Opname* is a variable then the call to current_op may be used to backtrack through operators, and in this case if either *Type* or *Priority* is an atom then only operators of that *Type* and/or *Priority* will be retrieved.

2.9 Modifying the Database

`assert(+`*Clause*`)`

Adds the specified *Clause* to the database as the last clause of its procedure.

`asserta(+`*Clause*`)`

Adds the specified *Clause* to the database as the first clause of its procedure.

`assertz(+`*Clause*`)`

Same as `assert/1`.

`assertx(+`*Clause*`,+`*Index*`)`

Adds the specified *Clause* to the database. If Index is positive then the clause is asserted as the *Index*th clause of its procedure. If *Index* is negative then the clause is asserted as the last but *Index*th of its procedure. If *Index* is zero, or too large a positive integer, then the clause is asserted as the last one. If *Index* is too large a negative number then the clause is asserted as the first one.

`retract(+`*Clause*`)`

If there is a clause in the data base that matches *Clause*, then that clause is deleted and any variables in *Clause* are instantiated by the match. On backtracking there is an attempt to find another matching clause. The search always starts at the beginning of the list of clauses for the relation name of *Clause*. The call fails when there is no matching clause.

`retractx(+`*Name*`,+`*Arity*`,+`*Index*`)`

If the dynamic predicate *Name/Arity* is defined, then one of its clauses, as specified by index, is deleted. If *Index* is positive, then the *Index*th clause of the procedure is deleted. If *Index* is negative then the last but *Index*th clause of the procedure is deleted. If *Index* is too large a positive integer then the last clause of the procedure is deleted. If *Index* is zero, or too large a negative number, then the first clause of the procedure is deleted.

`retractall(+`*Clause*`)`

All the data base clauses which match *Clause* are deleted, with any variables in *Clause* left uninstantiated. `retractall/1` always succeeds.

`abolish(+`*Pred*`)`

If predicate *Pred* is currently defined, then it becomes undefined. This can be used to remove definitions for both static and dynamic predicates.

`kill(+`*Pred*`)`

If predicate *Pred* is currently defined, then it becomes undefined. If predicate *Pred* is currently defined by a static segment, then all the predicates which are currently defined by that segment become undefined. This gives an easy method to remove all associated predicate definitions e.g. normally all predicates defined in the same file.

`save(+`*Pred*`,+`*File*`)`

Saves the object code for predicate(s) *Pred* to file *File*. *Pred* is either a single predicate or a list of predicates. Only those predicates which are defined are saved.

`save(+`*Pred*`,+`*File*`,+`*Mode*`)`

Opens the file *File* in *Mode* mode and saves the object code for predicate(s) *Pred*. *Pred* is either a single predicate or a list of predicates. Only those predicates which are defined are saved. *Mode* must be either `write` or `append`.

2.10 Property Management

`set_prop(+`*Object*`,+`*Property*`,?`*Value*`)`
: Sets a property value; *Value* becomes the remembered value of the property *Property* of the object *Object*. Any previously remembered value of the *Property* of the *Object* is lost.

`get_prop(+`*Object*`,+`*Property*`,-`*Value*`)`
: Retrieves the value of a property; *Value* is instantiated with the current value of the *Property* of *Object*. If there is no remembered value, the call fails.

`del_prop(+`*Object*`,+`*Property*`)`
: Removes a property value; the remembered *Property* value of *Object* is forgotten. A subsequent call to `get_prop(Object,Property,Value)` will fail. `del_prop/2` does nothing and succeeds if there is no such *Object* or *Property*.

`del_props(+`*Object*`)`
: Deletes all properties for a symbol; All the property-value pairs for the given *Object* are forgotten.

`del_cons(+`*Property*`)`
: Deletes all symbol values for a property. All the property-value pairs for the given *Property* are forgotten.

`remember(+`*Atom*`,?`*Value*`)`
: Stores *Value* as the currently stored value of *Atom*. Any previously assigned value is lost.

`recall(+`*Atom*`,-`*Value*`)`
: *Value* is instantiated with the currently stored value of atom. `recall/2` fails if *Atom* has no currently stored value.

`default_prop(+`*Atom*`,-`*Value*`,?`*Default*`)`
: *Value* is instantiated with the currently stored value of *Atom*, if there is one. It will be instantiated with *Default* if there is no currently stored value.

`forget(+`*Atom*`)`
: Deletes the stored value associated with *Atom*. A `recall/2` on *Atom* will now fail. A call to forget always succeeds, even if there is no stored value.

`get_cons(+`*Property*`,-`*List*`)`
: *List* is bound to to the list of all the atoms that currently have a value for the property *Property*.

`get_props(+`*Object*`,-`*List*`)`
: *List* is bound to the list of all the names of the properties currently associated with *Object*.

`list_props(+`*Object*`,-`*List*`)`
: *List* is bound to the list of all the 'Property=Value' pairs currently associated with *Object*.

2.11 Metalogical Primitives

`ntpl(+`*Len*`,-`*Tpl*`)`

Tpl is instantiated to the most general tuple having *Len* as its positive (`>`=1) arity, that is, a tuple with distinct variables for its functor and all of its arguments.

```
| ?- ntpl(3, Tpl).

Tpl = _125(_126,_127)
```

`tohollow(+`*Ground*`,-`*Hollow*`,+`*Varnames*`)`

Varnames a list of atoms. Only the atoms in the *Varnames* list of atoms are replaced by variables in the *Hollow* copy of *Ground*. Any atom can be on the *Varnames* list. The main use of this call is to convert a term and its associated list of variable names that has been read in using `gread/1`, `gread/2` or `gread/3` into a hollow term that can be used in an evaluation.

```
| ?- tohollow(foo(a,b,c), Hollow, [a,c]).

Hollow = foo(_175,b,_178)
```

`tohollow(+`*Ground*`,-`*Hollow*`,+`*Varnames*`,-`*Vars*`)`

Essentially the same as the above three argument use except that the list of variables in *Hollow* that have replaced the atoms on *Varnames* is also returned. The i*th* variable on *Vars* is the replacement for the i*th* variable name on *Varnames*. This is useful if you want to subsequently output bindings for these variables using the original variable names.

```
| ?- tohollow(foo(a,b,c), Hollow, [a,c], Vars).

Hollow = foo(_189,b,_192)
Vars = [_189,_192]
```

`toground(?`*Hollow*`,-`*Ground*`)`

Simple use to convert any term into a variable free term. *Ground* is bound to a copy of *Hollow* in which all the variables are replaced by new atoms beginning with underscores. Different variables are replaced by different underscore names, and there will be no clash with any atom that begins with underscore which already appears in *Hollow*.

```
| ?- toground(foo(A,B,C), Ground).

Unbound variables : A, B, C
Ground = foo('_0','_1','_2')
```

`toground(`*?Hollow*`,-`*Ground*`,-`*Varnames*`)`

Varnames will be bound to the list of all the underscore variable names that have been used to replace variables in *Hollow*.

```
| ?- toground(foo(A,B,C), Ground, Varnames).

Unbound variables : A, B, C
Ground = foo('_0','_1','_2')
Varnames = ['_0','_1','_2']
```

`toground(`*?Hollow*`,-`*Ground*`,-`*Vars*`,-`*Varnames*`)`

Same as the three argument use except that the corresponding list of the replaced variables is also returned as the binding for *Vars*.

If there are more variables in *Hollow* than given in *Vars*, new underscore names are given to the extra variables. More generally, the elements of *Vars* can be any terms. Non-variable terms on *Vars*, and the corresponding atoms on *Varnames* are ignored - but the lists must still be of the same length. This relaxation allows lists of names and variables returned by some previous use of `tohollow/3` or `tohollow/4` to be used as arguments to `toground/4` without having to remove variables that may have been bound by the intervening evaluation.

```
| ?- toground(foo(A,B,C), Ground, Vars, Varnames).

Unbound variables : A, B, C
Ground = foo('_0','_1','_2')
Vars = [A,B,C]
Varnames = ['_0','_1','_2']
```

`toground(`*?Hollow*`,-`*Ground*`,+`*Vars*`,+`*Varnames*`,-`*Usednames*`)`

Varnames a list of atoms, *Vars* a list of variables which may appear in *Hollow*. The *Varnames* list of atoms will be used to replace the variables of *Vars* in *Hollow* in the order in which they are given - the first variable of *Vars* is replaced by the first atom on the *Varnames* list, and so on.

Finally, *Usednames* is bound to the complete list of the actual variable names that have been used to replace variables in *Hollow*.

```
| ?- toground(foo(A,B,C),Ground,[A,B,C],['W','X','Y'],Usednames).

Unbound variables : A, B, C
Ground = foo(W,X,Y)
Usednames = [W,X,Y]
```

`varsin(`*?Term*`, -`*Vars*`)`

Vars is a list of the variables appearing in term *Term*

```
| ?- varsin((foo(X,Y) :- bar(X,Y,1,Z)), Vars).

Unbound variables : X, Y, Z
Vars = [X,Y,Z]
```

`forall(+`*Gen*`,+`*Test*`)`

This call succeeds if for all the solutions of *Gen*, *Test* is true.

```
| ?- forall(member(X, [1,2,1,3,5,2,-1]), X<3).

no
| ?- forall(member(X, [1,2,1,3,5,2,-1]), write(X)).
121352-1
Unbound variable : X
```

`findall(`*?Term*`,+`*Call*`,-`*List*`)`

List will be unified with a list of instantiations of *Term*, one for each successful evaluation of *Call*.

```
| ?- findall(X, (member(X, [1,2,1,3,5,2,-1]), X < 3), List).

Unbound variable : X
List = [1,2,1,2,-1]
```

The instantiations of term correspond to the different solution bindings for the variables in *Call*. At the end of the evaluation no variable in *Call* will be bound. All the 'local' variables in *Call*, the variables which do not appear in *Term*, or in any other condition in the clause or query in which the `findall/3` is used, are implicitly existentially quantified. All 'global' variables of *Call*, variables that are also used in other conditions in the clause or query, should be bound to variable free terms before the `findall/3` is evaluated.

Note that `findall/3` is faster then `bagof/3` (see below) and should be used instead of `bagof/3` when there will be no unbound global variables in the *Call* for which solutions are to be found.

```
| ?- findall(Y, (member(X, [1,2]), Y is X+1), List).

Unbound variables : X, Y
List = [2,3] ;

no
```

`bagof(`*?Eterm*`,+`*ExistentialCall*`,`*?List*`)`

ExistentialCall must be a term of the form *v1*`^`*v2*`^`...*vk*`^`*Call* where *Call* is a call term, and *v1*,...,*vk* (k>=0) are variables in *Call* (when k=0 just the *Call* is given). At the time of the call, *Call* will generally contain variables. The variables in *Call* that are not in either *Eterm* or the sequence *v1*,...,*vk* of variables preceding *Call* are the global variables of the `bagof/3` call. *v1*,...,*vk* are the existentially quantified variables of *Call*.

`bagof/3` partitions the list of all the values of *Eterm* for all the solutions of *Call* by different solution values for these global variables.

That is, suppose that in the space of all the successful evaluations of *Call* there are *n* different sets of bindings for its global variables. Then `bagof/3` will backtrack giving *n* different answers. Each answer will comprise a set of bindings for the global variables, and a corresponding value for *List* which comprises the instances of *Eterm* for different solutions of *Call* that make this assignment to the global variables.

```
| ?- bagof(Y, (member(X, [1,2]), Y is X+1), List).

Unbound variable : Y
X = 1
List = [2] ;

Unbound variable : Y
X = 2
List = [3] ;

no
| ?- bagof(Y, X^(member(X, [1,2]), Y is X+1), List).

Unbound variables : X, Y
List = [2,3] ;

no
```

`setof(`*?Eterm*`,+`*ExistentialCall*`,`*?List*`)`

Same as `bagof/3` except that the bindings for *List* will be ordered lists of terms without duplicates. The terms are ordered by the `@</2` primitive as a list of increasing terms.

```
| ?- bagof(Y, X^(member(X, [2,1,2,3,0,4]), Y is X+1), List).

Unbound variables : X, Y
List = [3,2,3,4,1,5]

| ?- setof(Y, X^(member(X, [2,1,2,3,0,4]), Y is X+1), List).

Unbound variables : X, Y
List = [1,2,3,4,5]
```

`map(+`*Rel*`,+`*Inlist*`)`

For each use of `map/2`, the term *Rel* may be either a relation name or a call term of the form reln(arg1, ..., argn) whose principal functor reln is a relation name.

If *Rel* is a unary relation name, this call will test if all the elements of a list *Inlist* satisfy *Rel*. If *Rel* is a compound term of the form `reln(`*Arg1*`,...,`*Argn*`)` then the call tests if all the elements *El* of the list *Inlist* satisfy `reln(Arg1,...,`*Argn*`,`*El*`)`.

`map(+`*Rel*`,?`*Inlist*`,?`*Outlist*`)`

If *Inlist* is given, `map/3` will produce a list *Outlist* such that each element of *Outlist* is in the relation *Rel* to the corresponding element of *Inlist*. Alternatively, `map/3` may be used to check that a given *Outlist* is in this relationship to *Inlist*.

`map(+`*Rel*`,+`*Inlist*`,+`*Invalue*`,?`*Outvalue*`)`

The given *Rel* is applied to 'accumulate' the elements of *Inlist* using *Invalue* as the initial value of the cumulated term. The variable *Outvalue* is bound to the final cumulated term.

`map(+`*Rel*`,+`*Inlist*`,+`*Outlist*`,+`*Invalue*`,?`*Outvalue*`)`

The given *Rel* is applied both to produce an *Outlist* from *Inlist*, and to 'accumulate' the elements of *Inlist* using *Invalue* as the initial value of the cumulated term.

2.12 Transformations on Reading

There are two forms of program transformations which can be done by ICP when reading in files.

The first is using `term_expansion/2` is based on a clause by clause transformation and is primarily used for converting grammar rules. The implementation of grammar rules is the same as that of Quintus Prolog *Quintus Prolog Development Environment*, Quintus Computer Systems, Inc., Mountain View, California, USA. The grammar rule translator used is based on a public domain program written by Richard O'Keefe.

The second method allows users to define a complete transformation on a file determined by the file name extension.

2.12.1 Definite Clause Grammars

A grammar rule takes the form

```
head --> body
```

`expand_term(+`*Term1*`, -`*Term2*`)`
: This predicate is used by `consult/1` and `compile/1` to determine the transformation of terms before they are asserted or compiled. *Term1* is the input term and *Term2* the output term. By default, only terms corresponding to grammar rules are transormed; in all other cases *Term2* is identical to *Term1*. This performance may be changed if the user predicate `term_expansion/2` is defined.

`phrase(+`*Phrase*`, ?`*List*`)`
: This predicate determines whether a list *List* is a phrase of type *Phrase*

`phrase(+`*Phrase*`, ?`*List*`, ?`*Rest*`)`

`'C'(?`*S1*`, ?`*Terminal*`, ?`*S2*`)`
: This predicate is used in expessing grammar rules and is defined by the clause

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

`term_expansion(+`*Term1*`, -`*Term2*`)`
: The standard transformation of terms before compilation or consultation can be overridden by defining this predicate. It can be used to define different 'sugared' syntaxes.

`query_expansion(+`*Term1*`, -`*Term2*`)`
: A user query may be transformed before calling the goal by defining this predicate. If it succeeds, *Term2* will be executed but the variable bindings will be printed as usual upon completion.

2.12.2 File Transformations

The transformations in the previous section are at the clause level. A more powerful method of transformation is at the level of files, since this allows contextual transformation of clauses. For example, when writing a preprocessor for an object-oriented extension to Prolog, we may wish to generate default inheritance clauses for objects depending on whether the object has an explicit

superclass defined in the same file. This requires a global view of all the clauses in a file rather than the simple clause-level transformations.

File-level transformations are specified by defining a new file extension type and a user hook relation to read (and transform) the new file type. The user hook relation is either `user_consult/3` or `user_compile/3`. An example of the use of these predicates is shown in '`$ICP_INSTALLDIR/skilaki/source/icp_comp.pl`'.

`user_file_type(`*?Ext*`, `*?Type*`)`
: This declaration allows the user to define an alternative compilation or consultation for files with the extension *Ext*. The term *Type* may be used to select which paticular transformations to perform on reading the file.

`read_prolog_file(-`*Preds*`)`
: This predicate reads an entire file using any user-defined hooks, returning the predicates in the list *Preds*.

`user_consult(+`*Type*`, +`*FileName*`, -`*Preds*`)`
: On consultation, if a `user_file_type/2` declaration exists for the file, the system calls this predicate having already opened the file *FileName* and set current input to it. This should read and process the file returning a list of predicates as specified below. These predicates will then be asserted into the database automatically.

  The file may be read using `read_prolog_file/1` and post-processed into the list *Preds*. The intended usage is to allow more flexible transformations than grammer rules would allow.

```
user_file_type('.foo', foo_type).

user_consult(foo_type, File, Preds) :-
read_prolog_file(Source),
process(Source, Preds).
```

`user_compile(+`*Type*`, +`*FileName*`, -`*Preds*`)`
: As `user_consult/3` but *Preds* are automatically compiled rather than asserted.

In all the above predicates, *Preds* is a list of terms of the form `pr(`*Pred*`/`*Arity*`,`*List*`)` where *Pred* is the predicate name (an atom), *Arity* is an integer and *List* is a list representing the clauses for this predicate. Each term in this list is of the form *VarNames*`-`*GroundClause*, where *VarNames* is a list of atoms representing the variable names in the ground clause *GroundClause*.

2.13 List Handling

`qsort(+`*List*`,-`*Sorted*`,+`*Test*`)`
: Test must be the name of a binary relation. The call succeeds if *Sorted* can be unified with the result of sorting *List* with respect to *Test*.

`append(?`*List1*`,?`*List2*`,?`*List3*`)`
: Succeeds if *List3* is *List1* followed by *List2*.

`member(?`*El*`,?`*List*`)`
: Succeeds if *El* is a member of list *List*. This may be used to backtrack over the whole list.

`memberchk(?`*El*`,?`*List*`)`
: Succeeds if *El* is a member of list *List*. This is different from `member/2` since it does not backtrack.

`on(?`*El*`,?`*List*`)`
: Same as `member/2`

`occ(?`*X*`,?`*L*`)`
: Succeeds if term *X* occurs in unsafe list *L*. An unsafe list is one which has a variable as the tail. If *X* is not in *L* then it is put there.

`no_occ(?`*X*`,?`*L*`)`
: Succeeds if term *X* is not in unsafe list *L*.

`remove(?`*Item*`,-`*List*`,-`*Remainder*`)`
: Succeeds if *Remainder* is the list *List* with the element *Item* removed. If *Item* does not occur in *List*, the call to remove fails. If *Item* occurs in *List* more than once, subsequent occurrences will be removed on backtracking.

`delete(?`*Item*`,-`*List*`,-`*Remainder*`)`
: Same as `remove/2`

`reverse(?`*List*`,?`*Revlist*`)`
: Succeeds if *Revlist* is the list *List* with the order of its elements reversed.

2.14 String and Atom Handling

`name(?`*X*`,?`*L*`)`
: *X* is that atom or integer such that *L* is a list of the ASCII codes for its printed representation.

```
| ?- name(hello, List), name(Hello, List).

List = [104,101,108,108,111]
Hello = hello

| ?- A1='12', name(A1, List), name(A2, List).

A1 = '12'
List = [49,50]
A2 = 12

| ?- A1='12', name(A1, List), name(A2, List), A1=A2.

no

| ?- A1=12, name(A1, List), name(A2, List).

A1 = A2 = 12
List = [49,50]
```

`atom_chars(`*?X*`,`*?L*`)`

X is the atom such that *L* is a list of the ASCII codes for its printed representation.

```
| ?- A1='12', atom_chars(A1, List), atom_chars(A2, List).

A1 = A2 = '12'
List = [49,50]

| ?- A1='12', atom_chars(A1, List), atom_chars(A2, List), A1=A2.

A1 = A2 = '12'
List = [49,50]

| ?- A1=12, atom_chars(A1, List).

no
```

`number_chars(`*?X*`,`*?L*`)`

X is the number such that *L* is a list of the ASCII codes for its printed representation.

```
| ?- A1='12', number_chars(A1, List).

no

| ?- A1=12, number_chars(A1, List), number_chars(A2, List).

A1 = A2 = 12
List = [49,50]
```

`numbervars(`*?Term*`, +`*Start*`, -`*End*`)`

Transforms each of the variables in *Term* into ground terms of the form `'$Var'(`*N*`)` where *N* is an integer greater than or equal to the integer *Start*.

```
| ?- numbervars(a(X,Y), 0, End), display(a(X,Y)).
a($VAR(0),$VAR(1))
X = A
Y = B
End = 2
```

The first variable in *Term* gets *N* equal to *Start* while the second distinct variable gets *Start*+1 and so on. *End* is the value of *N* when the last variable has been grounded.

`concat(`*?X*`,`*?Y*`,`*?Z*`)`

Z is instantiated to the atom formed by concatenating the printed representation of *X* with that of *Y*. The limit of the size of a single atom is 255 characters. If you try to construct a new atom with more than 255 characters the extra trailing characters will simply be ignored.

```
| ?- concat(foo, bar, Full).

Full = foobar

| ?- concat(Start, bar, foobar).

Start = foo

| ?- concat(foo, End, foobar).

End = bar
```

`concat_atom(`*?List*`,`*?Atom*`)`

This primitive concatenates a list of constants *?List* into a single atom *?Atom*. The constants may be numbers or atoms.

`pname(`*?T*`,`*?N*`)`

This primitive converts between a Prolog term and an atom representing its print name. If *T* is bound, *N* is bound to the atom of the print name of *T*.

If N is bound, T is bound to the term whose print name is N.

```
| ?- pname(a(X, 2), N).

Unbound variable : X
N = 'a(_0,2)'

| ?- pname(T, 'a(X, 2)').

T = a(_243,2)
```

`pname(`*?T*`,`*?N*`,`*?VNames*`,`*?Vars*`)`

Same as `pname/2` but with better control of the print name of variables. If *Vnames* is given is a variable, N is bound to the atom of the print name of T, with each variable in list *Vars* being represented by the corresponding atom in list *VNames*. In the second form, T is bound to the term whose print name is N, *Vars* is bound to a list of variables found in the term and *VNames* is bound to a list of the corresponding print names.

```
| ?- pname(a(X, 2), N, ['V1'], [X]).

Unbound variable : X
N = 'a(V1,2)'

| ?- pname(T, 'a(X, 2)', VNames, Vars).

T = a(_273,2)
VNames = ['X']
Vars = [_273]
```

This predicate is useful for meta-programming.

`copy_term(`*?Term*`, `*?CopyTerm*`)`

Creates a copy of term *Term* with new variables but preserving bindings

```
| ?- copy_term(a(X, b(G,X)), Copy).

Unbound variables : G, X
Copy = a(_141,b(_146,_141))
```

`gensym(+`*Prefix*`, -`*Atom*`)`

Creates an atom *Atom* whose prefix is *Prefix* and whose suffix is an integer. Successive calls to `gensym/2` are guaranteed to give different atoms.

```
| ?- gensym(a,Atom1), gensym(a,Atom2), gensym(b, Atom3).

Atom1 = a1
Atom2 = a2
Atom3 = b1
```

2.15 Threads

In the description below, a pipe is a communication channel between two threads or processes. A pipe has two end which are called ports. An output port is where terms enter the pipe (i.e. written out by a thread), and an input port is where the terms leave the pipe (i.e. read in by a thread).

`fork(+`*Goal*`)`
: Forks a child thread to run *Goal*. The current thread is not suspended. The child thread must wait for a `resume` call or a time slice to start executing.

`fork(+`*Goal*`,-`*Th*`)`
: Creates a child thread to run *Goal*, a thread identifier *Th* is returned. The current thread is not suspended. The child thread must wait for a `resume` call or a time slice to start executing.

`qprolog(+`*Goal*`)`
: Similar functionality to `fork/1`, but without the overhead of creating a new thread. The *Goal* is executed in a background reserved thread. Use this in preference to `fork/1` only when *Goal* is simple, since complex queries will delay other calls to `qprolog/1`.

`new_thread(-`*Th*`)`
: This creates a new thread structure and binds *Th* to an integer identifier for the thread. This low-level primitive is typically not used by the user directly.

`thread(?`*Th*`)`
: Returns the identifier of the current thread.

`kill_thread(+`*Th*`)`
: Kills the specified thread and reclaims the memory space.

`suspend`
: Sets the status of the current thread to unrunnable. Note that this does not immediately suspend the thread, but merely removes it from the run queue. The thread must be explicitly named in order to resume it.

`suspend(+`*Th*`)`
: Sets the status of the thread *Th* to unrunnable. This removes the specified thread from the run queue. The thread must be explicitly resumed using `resume/1` to continue executing.

`resume` Non-specific scheduler. The next thread in the circular run queue resumes execution.

`resume(+`*Th*`)`
: Resume execution of the specified thread. This is the only way to change the status of a thread from unrunnable to runnable. If *Th* is 0, the next runnable thread is resumed. `resume/1` returns an error if *Th* is not a currently valid thread.

`timeslice(+`*N*`)`
: If N is the integer 0, time-slicing is switched off. This has the effect of preventing other runnable threads from executing, thus giving the current thread exclusive use of the processor. If N is any other integer, time-slicing is resumed. This primitive is used when atomicity is required or when critical code regions are entered.

`pipe(-`*OP*`, -`*IP*`)`
: Creates a new pipe and binds *OP* to the output end and *IP* to the input end of the pipe. Terms which are written to *OP* may be read from *IP*. The structure of a port term is `port(`*N*`)` where *N* is an integer.

`write_pipe(+`*OP*`, ?`*T*`)`
: Outputs the term *T* to the output port *OP*. *OP* must be an output port owned by the current thread, otherwise the primitive fails. Ownership of a port is automatically set to the current thread if there is no current owner. Note that creation of a pipe does not set ownership for the ports. Thus a pipe may be created by one thread and used by another thread. If any thread is suspended on a read, `write_pipe/2` wakes it after *T* is written out to the pipe.

`look_pipe(+`*IP*`, -`*T*`)`
: Copies a term *T* from the input port *IP*. The term is not removed from the pipe. *IP* must be an input port owned by the current thread, otherwise the primitive fails. Ownership of a port is automatically set to the current thread if there is no current owner. The pipe is 'locked' before the term is read. This is to prevent multiple readers accessing the pipe simultaneously. The lock is not released by this primitive (see `commit_read/1`). `look_pipe/2` suspends if the port is already locked or if the pipe is empty. If the output port of the pipe is closed, `look_pipe/2` binds *T* to the constant `end_of_file`.

`commit_read(+`*IP*`)`
: Removes the term previously read by `look_pipe/2` from the pipe and unlock the pipe. *IP* must be an input port owned by the current thread, otherwise the primitive fails. `commit_read/1` fails if the pipe is not locked (i.e. no previous `look_pipe/2`).

`read_pipe(+`*IP*`, ?`*T*`)`
: Reads a term *T* from the input port *IP*. Semantically, this is equivalent to `look_pipe(`*IP*`,`*T*`)` followed sequentially by `commit_read(`*IP*`)`.

`empty_pipe(+`*IP*`)`
: Is true if pipe *IP* has no data to be read. %LOOK

`is_iport(+`*Port*`)`
: Is true if *Port* is an input port.

`is_oport(+`*Port*`)`
: Is true if *Port* is an output port.

`unlock(+`*Port*`)`
: Unlocks a pipe which has been locked by a previous `look_pipe/2`. The port must be owned by the current thread, otherwise the primitive fails. This is used in the case where the term read is not the one expected. The pipe is unlocked so that another thread can re-read the same term.

`close_port(+`*Port*`)`
: Closes the specified port. The port must be owned by the current thread, otherwise the primitive fails. `close_port/1` suspends if the pipe is locked and fails if the port has a suspended operation pending. If an output port is closed, subsequent (and suspended) reads from the input port will return `end_of_file`. Closing an input port does not affect the output port. If both ports are closed, space used for the pipe is reclaimed.

`release_port(+`*Port*`)`
: Releases ownership of the port to allow other threads to use it. The port must be owned by the current thread, otherwise the primitive fails. `release_port/1` suspends if the pipe is locked.

`shell` This starts a read-eval-print loop in a new window to handle Prolog queries. It is usually used as the goal to be executed in a new forked thread.

`exit` Kills the current thread and resumes another runnable thread, if any. This is usually used to exit a shell as started above a la Unix.

2.16 Prolog - Parlog Interface

`xparlog` Starts the parlog thread and redirects its console input and output to a new window under X11. It also opens a pipe for communication between the prolog and parlog processes. The new window may be used to type parlog queries directly or you may use `parlog/1` to call it from prolog. Only one parlog thread may be started.

`parlog` Switches the query handler from prolog to parlog. The first time that `parlog/0` is called, a parlog thread is started and a pipe is opened for communication between the prolog and parlog processes. Subsequent calls just switch to the existing parlog thread.

This primitive is useful when X Windows is not available or if you wish to run ICP in an Emacs buffer. To get back to prolog, type the parlog query `prolog/0`.

`parlog(+`*Goal*`)`
: Passes a goal *Goal* to the parlog thread to solve.

`close_parlog`
: Closes the communication port to parlog.

2.17 Miscellaneous Primitives

`unix(+`*A*`)` Execute the atom *A* as a unix command.

`cd(+`*A*`)` Change current directory to *A*.

`cd` Change current directory to the user's home directory.

`ls` Execute the unix command '`ls`' to list the current directory.

`ls(`*Dir*`)` Execute the unix command '`ls`' to list the directory *Dir*.

`pwd` Display the path name of the current directory.

`edit(+`*File*`, +`*Editor*`)`
: Edit the file *File* using the editor, *Editor*. If the filename does not have an extension, the suffix "`.pl`" will be added.

`edit(+`*File*`)`
: Edit the file *File* using the editor, stored as property `editor` (by default '`vi`'). If the filename does not have an extension, the suffix "`.pl`" will be added.

`vi(+`*File*`)` Edit the file *File* using the '`vi`' editor.

`out_of_date(+`*File*`, Type)`
: Is true if the file *File*('`.icp`') is older than file *File*(.*Type*). The files are can exist anywhere on the current search path.

`max(+`*I*`,+`*J*`,-`*K*`)`
: *K* is bound to the greater of *I* and *J*.

`arb_member(?`*El*`,?`*Term*`)`
: Succeeds if *El* is a subterm of *Term*. This is a deterministic test.

`not_arb_member(?`*El*`,?`*Term*`)`
: Succeeds if *El* is not a subterm of *Term*.

`address(?`*V*`,-`*Address*`)`
: Binds *Address* to the heap-address of variable *V*.

`time(-`*Time*`)`
: Binds *Time* to CPU time (in 60ths of a second) since last call.

`realtime(-`*Time*`)`
: Binds *Time* to the time since 00:00:00 GMT, Jan. 1, 1970 measured in seconds (the Unix epoch).

`ctime(`*?Time, ?Year, ?Month, ?Day, ?Hour, ?Minutes, ?Seconds*`)`
: If *Time* is a number (as returned by realtime/1), binds the remaining arguments with the corresponding integer representations. Alternatively if the other arguments are integers, instantiates *Time*.

`statistics(`*?Keyword*`, -`*List*`)`
: This gives various execution statistics. These statistics pertain to the thread in which the call is issued. For each of the values of *Keyword*, *List* is unified with a list of values.

  `runtime` `List = [`*Time1*`,`*Time2*`]` where the times are since the start of prolog execution and since the last call to `statistics/2` with *Keyword* = `runtime`. Both times are in milliseconds.

  `garbage_collection`
  : `List = [`*Number*`,`*Bytes*`,`*Time*`]` where `Number` is the number of garbage collections, *Bytes* the total number of bytes collected and *Time* the time, in milliseconds, spent doing the garbage collection.

`errno(`*?Number*`, -`*Message*`)`
: If *Number* is a variable, instantiates *Number* to the 'C' error number (`errno`) of the underlying system. *Message* is the textual message for the error.

  If *Number* is a number, *Message* is the textual message for the error corresponding to this number.

`getenv(+`*EnvName*`, -`*Value*`)`
: Gets the value *Value* of the Unix environment variable *EnvName*.

`debugicp` Drops into the WAM-level ICP debugger.

`noref` Displays on `user_error` a listing which gives the name and arity of the currently referenced but undefined static predicates.

`window_label(+`*Name*`)`
: Sets the name in the xterm title bar to *Name*.

`icon_label(+`*Name*`)`
: Sets the name in the xterm icon to *Name*.

`get_prolog_flag(+`*Flag*`, -`*Value*`)`
: Gets the value, *Value*, of the flag *Flag* for the thread in which the goal appears. The currently supported flags are

`debug` The values can be one of the atoms `on`, `off`

`undefined_predicate`
The values can be one of `fail` `warning` `error`. The default is `error`.

`set_prolog_flag(+`*Flag*`, -`*Value*`)`
Sets the value, *Value*, of the flag *Flag* for the thread in which the goal appears. The currently supported flags are

`argc(-`*Number*`)`
The number of arguments passed to prolog at initialization. These must be prefixed by `-z`.

`argv(-`*List*`)`
The list of arguments passed to prolog at initialization. These must be prefixed by `-z`.

`cursor(+`*Stream*`, ?`*Pos*`)`
If *Pos* is a variable, this returns the position in the stream *Stream*. If *Pos* is an integer, the stream position is set to its value.

`shell_escape`
Prompts the user for a goal to be executed. This is a useful menu option to provide so that queries can be made from deep inside a user program.

`get_path(-`*PathList*`)`
This predicate returns a list of the directories which will be searched when prolog is trying to locate a file.

`set_path(+`*PathList*`)`
This predicate can be used to set the directories which will be searched when prolog is trying to locate a file. *PathList* is a list of directories, each of which is an atom.

3 TCP Interface

To allow for the communication between different ICP processes, a TCP interface is provided. To make the interface accessible to users who are not experts at TCP and or socket level programming, we provide a restricted subset of the full TCP functionality provided at the C level. The intention was that simple communication can be written quickly without knowledge of the underlying communication protocol.

The primitives described here are implemented for both the Prolog and Parlog sub-systems in ICP. The TCP interface is not part of the core ICP system and must be loaded by a command of the form

```
?- ensure_loaded(tcp).
```

The Prolog and Parlog interfaces are distinct libraries and so must be loaded separately.

3.1 Introduction to TCP/IP

It is beyond the scope of this manual to give a full description of the TCP/IP communication protocol. For details the reader is refered to *Unix Network Programming* by W. Richard Stevens and to the Unix documentation.

In TCP communications happen through *sockets* which are similar to file handles used in file I/O. Inter-process communication occurs between two *peers*; the source and destination addresses. An address consists of two components

network address
: This specifies the IP address of the machine

port
: This allows multiple peers on a given machine

There are essentially two means of communication supported under the TCP protocol, *connection* and *connectionless*.

3.2 Connection Oriented Protocol

A socket is created and *bound* to a particular *port* and *host address*. The host address indicates the machine with which to communicate. Typically a single process on any one machine will listen for data on a particular port. Once a socket has been bound, subsequent reads on this socket do not require the specification of the source of the data (tcp_send/2, tcp_send/3). A connected server will typically have the following form

```
...,
tcp_server(Port, Socket),           % open and bind socket
tcp_accept(Socket, NewSocket),      % accept connection
tcp_recv(NewSocket, Term1),         % read from connection
...,
tcp_send(NewSocket, Term2),         % write to connection
...,
tcp_close(NewSocket),               % close connection
...,                                % here could do more accepts
tcp_close(Socket).                  % close socket
```

The corresponding clients which connect to the server would have the following form

```
...,
tcp_client(Port, Address, Socket),  % connect to server address and port
tcp_send(Socket, Term1),            % send data to server
...,
tcp_recv(Socket, Term2),            % receive data from server
...,
tcp_close(Socket).
```

`tcp_server(+`*Port*`, -`*Socket*`)`
`tcp_server(+`*Port*`, -`*Socket*`, +`*Address*`)`

This predicate may be used to initiate a connection-oriented server. It is defined as

```
tcp_server(Port, Socket) :-
        tcp_open(connection, Socket),
        tcp_bind(Socket, Port),
        tcp_listen(Socket).
```

The three argument form passes the extra argument to `tcp_bind/3`.

`tcp_client(+`*Port*`, +`*Address*`, -`*Socket*`)`
`tcp_client(+`*Port*`, +`*Address*`, -`*Socket*`, +`*TimeOut*`)`

This predicate may be used to initiate a connection-oriented client. It is defined as

```
tcp_client(Port, Address, Socket, TimeOut) :-
        tcp_open(connection, Socket),
        tcp_connect(Socket, Port, Address, TimeOut).
```

See `tcp_connect/4` for details about the arguments.

`tcp_accept(+`*Socket*`, -`*NewSocket*`)`
`tcp_accept(+`*Socket*`, -`*NewSocket*`, -`*Port*`, -`*Address*`)`
`tcp_accept(+`*Socket*`, -`*NewSocket*`, -`*Port*`, -`*Address*`, +`*TimeOut*`)`
In the case of the server in a connection oriented protocol after listening, the socket must accept connections from clients. This returns a new socket *NewSocket* through which communication with the client specified by *Address* and *Port* can proceed. This is equivalent to the C fragment

```
{
        struct sockaddr_in add;
        int socket=Socket, newsocket=NewSocket;
        int length=sizeof(struct sockaddr_in);

        newsocket = accept(socket,(struct sockaddr *) &add,
                &length);
        Address = ntohl(add.sin_addr.s_addr);
        Port = ntohs(add.sin_port);
}
```

The variable *TimeOut* must have one of the values

`block` The call will block until a connection is established

`poll` This throws error 702 if immediate connection is not possible. By default this error simply fails but a user defined error handler can override this.

Integer The call will block for *Integer* seconds after which if no connection is established error 702 is thrown.

`tcp_send(+`*Socket*`, +`*Term*`)`
`tcp_send(+`*Socket*`, +`*Term*`, +`*EFlag*`)`
This is used to send data on a connected socket. It can not be used for connectionless protocols as it does not specify the address of the recipient. *Eflag* must be one of the following atoms

`normal` Normal data in encoded form

`raw` Normal data. Suitable for communication with non prolog processes. Raw data must be a prolog atom.

`tcp_recv(+`*Socket*`, -`*Term*`)`
`tcp_recv(+`*Socket*`, -`*Term*`, +`*EFlag*`)`
`tcp_recv(+`*Socket*`, -`*Term*`, +`*EFlag*`, +`*TimeOut*`)`
Is used to receive data from a socket. The variable `+`*Timeout* must have one of the values

`block` The call will block until a connection is established

`poll` This throws error 702 if immediate connection is not possible. By default this error simply fails but a user defined error handler can override this.

Integer The call will block for *Integer* seconds after which if no connection is established error 702 is thrown.

`tcp_recv/2` is equivalent to the following definition

```
tcp_recv(Socket, Term) :-
        tcp_recv(Socket, Term, normal, block).
```

Eflag must be one of the following atoms

`normal` Normal data in encoded form

`raw` Normal data. Suitable for communication with non prolog processes. Raw data must be a prolog atom.

`peek` Used to determine if there is normal data without removing it from the buffer.

`peek_raw` Used to determine if there is raw data without removing it from the buffer.

3.3 Connectionless Oriented Protocol

Here the socket is not associated with any particular port or host address. Hence to send data, the destination port and host address must be specified (tcp_sendto/4, tcp_sendto/5)

An example of a connectionless server follows

```
tcp_connectionless(ServerPort, Socket),        % open socket
tcp_recvfrom(Socket, Term1, Port, Machine),    % receive data from client
...,
tcp_sendto(Socket, Term2, Port, Machine),      % send data to client
...,
tcp_close(Socket).

tcp_connectionless(ClientPort, Socket),
tcp_sendto(Socket, Term1, Port, Machine),      % send data to server
...,
tcp_recvfrom(Socket, Term2, Port, Machine),    % receive data from server
...,
tcp_close(Socket).
```

`tcp_connectionless(+`*Port*`, -`*Socket*`)`

This predicate may be used to initiate a connectionless server or client. It is defined as

```
tcp_connectionless(Port, Socket) :-
        tcp_open(connectionless, Socket),
        tcp_bind(Socket, Port).
```

`tcp_broadcast(+`*Port*`, -`*Socket*`)`

This predicate may be used to initiate a broadcast socket. It is defined as

```
tcp_broadcast(Port, Socket) :-
        tcp_open(broadcast, Socket),
        tcp_bind(Socket, Port).
```

Typically, *Port* is specified as 0, which lets the system choose an unused port number.

`tcp_sendto(+`*Socket*`, +`*Term*`, +`*Port*`, +`*Address*`)`
`tcp_sendto(+`*Socket*`, +`*Term*`, +`*Port*`, +`*Address*`, +`*EFlag*`)`

This is used to send data on either a connected or connectionless socket.

Eflag must be one of the following atoms

`normal` Normal data in encoded form

`raw` Normal data. Suitable for communication with non prolog processes. Raw data must be a prolog atom.

`tcp_sendbr(+`*Socket*`, +`*Term*`, +`*Port*`)`
`tcp_sendbr(+`*Socket*`, +`*Term*`, +`*Port*`, +`*EFlag*`)`

This broadcasts on the specified port. The socket must have been opened in broadcast mode.

Eflag must be one of the following atoms

`normal` Normal data in encoded form

`raw` Normal data. Suitable for communication with non prolog processes. Raw data must be a prolog atom.

`tcp_recvfrom(+`*Socket*`, -`*Term*`, -`*Port*`, -`*Address*`)`
`tcp_recvfrom(+`*Socket*`, -`*Term*`, -`*Port*`, -`*Address*`, +`*EFlag*`)`
`tcp_recvfrom(+`*Socket*`, -`*Term*`, -`*Port*`, -`*Address*`, +`*EFlag*`, +`*TimeOut*`)`

Is used to receive data from a socket, also returning the IP address and port of the sender. The variable `+`*Timeout* must have one of the values

`block` The call will block until a connection is established

`poll` This throws error 702 if immediate connection is not possible. By default this error simply fails but a user defined error handler can override this.

Integer The call will block for *Integer* seconds after which if no connection is established error 702 is thrown.

`tcp_recvfrom/4` is equivalent to the following definition

```
tcp_recvfrom(Socket, Term, Port, Address) :-
        tcp_recvfrom(Socket, Term, Port, Address, normal, block).
```

Eflag must be one of the following atoms

`normal`	Normal data in encoded form
`raw`	Normal data. Suitable for communication with non prolog processes. Raw data must be a prolog atom.
`peek`	Used to determine if there is data without removing it from the buffer.

3.4 Miscellaneous TCP Predicates

This section describes some useful TCP primitives which are not restricted to either connection or connectionless protocols. The are typically used to enquire about the state of the communication channel.

`tcp_close(+`*Socket*`)`
: This predicate is used to close a socket obtained with one of the calls `tcp_socket/3`, `tcp_open/2` or `tcp_accept/5`.

`tcp_checkconn(+`*Socket*`)`
: This predicate succeeds if the socket is ready to be written to and fails if it is not.

`tcp_checkrecv(+`*Socket*`)`
`tcp_checkrecv(+`*Socket*`, +`*EFlag*`)`
: This predicate succeeds if the socket is ready to be read from and fails if it is not.

`tcp_getsockaddr(+`*Socket*`, -`*Port*`, -`*Address*`)`
: This predicate returns the IP address *Address* and port number of this side of a socket.

```
| ?- tcp_server(5569, Socket).

Socket = 0

| ?- tcp_getsockaddr(0, Port, Address).

Port = 5569
Address = 2460554497

| ?- tcp_gethost(Name, Address).

Name = laotzu
Address = 2460554497
```

`tcp_getpeeraddr(+`*Socket*`, -`*Port*`, -`*Address*`)`

This predicate returns the IP address *Address* and port number of the far side of a connected socket.

```
| ?- tcp_server(5569, Socket).

Socket = 0

| ?- tcp_accept(1, NewSocket, NewPort, Address).

NewSocket = 1
NewPort = 1108
Address = 2460554499

| ?- tcp_getpeeraddr(1, Port, Address).

Port = 1108
Address = 2460554499
```

`tcp_currenthost(-`*Name*`, -`*Address*`)`

Gets the name and address of the current host.

```
| ?- tcp_currenthost(Name, Address).

Name = sophia
Address = 2460554499
```

`tcp_gethost(?`*Name*`, ?`*Address*`)`

The machine of name *Name* has IP address *Address*. The predicate is deterministic.

```
| ?- tcp_gethost(laotzu, Address).

Address = 2460554497

| ?- tcp_gethost(Name, 2460554497).

Name = laotzu

| ?- tcp_gethost('146.169.21.1', L).

L = 2460554497
```

`tcp_getport(?Name, ?Protocol, ?Port)`

The named port *Name*, is on port *Port*, using protocol *Protocol*. These values correspond to the entries in file `/etc/services`. Valid modes of use are `tcp_getport(+Name,?Protocol, ?Port)` and `tcp_getport(?Name, ?Protocol, +Port)`.

```
| ?- tcp_getport(login, Protocol, Port).

Protocol = tcp
Port = 513

| ?- tcp_getport(Name, Protocol, 513).

Name = who
Protocol = udp

| ?- tcp_getport(Name, tcp, 513).

Name = login
```

The predicate is deterministic; It only finds one port per name.

`tcp_real_socket(+Socket, -Handle)`

Predicates which return socket identifiers (such as `tcp_open/2` return integers `>=` 0 which are unique identifiers for the socket. Given a socked identifier *Socket*, `tcp_real_socket/2` returns the file handle for the socket. This is useful if user defined predicates require this descriptor.

3.5 Low Level TCP primitives

This section describes other less frequently used TCP primitives. They are provided here for completeness.

`tcp_open(+Type, -Socket)`

The predicate `tcp_open/2` is defined in terms of `tcp_socket/3` providing a higher level method of openning sockets. Opens a TCP connection of type *Type* returning the identifier *Socket*. *Type* is one of the following atoms

`connection`

A connected socket is opened. This provides sequenced, reliable two-way connection based byte streams. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a `tcp_connect/4` call. Once connected, data may

be transferred using `tcp_send/3` and `tcp_recv/4` calls. When a session has been completed a `close/1` call may be performed. This is equivalent to the C call

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
```

and is defined as

```
tcp_open(connection, Socket) :-
        tcp_socket(sock_stream, ipproto_ip, Socket).
```

`connectionless`

A connectionless socket is opened. This provides unreliable messages of a fixed (typically small) maximum length. Data may be transferred using `tcp_sendto/5` and `tcp_recvfrom/5` calls. This is equivalent to the C call

```
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
```

and is defined as

```
tcp_open(connectionless, Socket) :-
        tcp_socket(sock_dgram, ipproto_ip, Socket).
```

`broadcast`

A broadcast socket is opened. This is a connectionless socket intended for broadcasting messages to the local network. Messages written to such a socket may be received by any machines listening on the local network. These are connectionless, unreliable messages of a fixed (typically small) maximum length. The messages will be delivered to all broadcast endpoints in the network. Data may be transferred using `tcp_send/3` and `tcp_recvfrom/5` calls. This is equivalent to the C fragment

```
{
        int i = 1;
        sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
        setsockopt(sock,SOL_SOCKET,SO_BROADCAST,&i,
                sizeof(int));
}
```

and is defined as

```
tcp_open(broadcast, Socket) :-
        tcp_socket(sock_dgram, ipproto_ip, Socket),
        tcp_setsockopt(Socket, so_broadcast, 1).
```

`tcp_bind(+`*Socket*`, +`*Port*`)`
`tcp_bind(+`*Socket*`, +`*Port*`, +`*Address*`)`

Once a socket has been opened by a call to `tcp_open/2` or `tcp_socket/3`, and with the exception of a connected client, it should be bound to a port and an IP address. Some ports - typically 1-1023 are reserved by the system. If *Port*=0 the system will assign an available port. The predicate `tcp_getport/3` may be used to obtain the port number of a port named in `/etc/services`. This is an interface to `tcp_bind/3` and is defined as

```
tcp_bind(Socket, Port) :- tcp_bind(Socket, Port, inaddr_any).
```

`tcp_bind/3` is as `tcp_bind/2` except that the variable *Address* is either the atom `inaddr_any` or an IP address (integer such as 2460554498, or the equivalent atom in dot notation '146.169.21.2') of an ethernet interface of the machine (obtained from the predicate `tcp_gethost/2`). This is only important for machines which have more than one ethernet interface. Specifying an IP address means that only messages on that network will be sent/received. If *Address* is `inaddr_any` then all the interfaces are used by default. This is equivalent to the following C fragment.

```
{
        struct sockaddr_in server;
        u_long machine = Address
        u_short port = Port
        int socket = Socket;

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(machine);
        server.sin_port = htons(port);
        bind(socket, (struct sockaddr *) &server, sizeof(server));
}
```

`tcp_connect(+`*Socket*`, +`*Port*`, +`*Address*`)`
`tcp_connect(+`*Socket*`, +`*Port*`, +`*Address*`, +`*TimeOut*`)`

In the case of the client in a connection oriented protocol the socket needs to be connected to the server. The *Address* may be specified as a machine name (e.g. confucius, or 'confucius.doc.ic.ac.uk'), or the IP address in integer form (e.g. 2460554498) or an atom in dot notation form (e.g. '146.169.21.2').

`tcp_connect/4` is equivalent to the C fragment

```
{
        struct sockaddr_in add;
        u_long machine=Address;
        u_short port=Port;
        int socket=Socket;

        add.sin_family = AF_INET;
        add.sin_addr.s_addr = htonl(machine);
        add.sin_port = htons(port);
        fcntl(socket, F_SETFL, FASYNC);
        connect(socket, (struct sockaddr *) &add, sizeof(add));
        fcntl(socket, F_SETFL, 0);
}
```

The variable *TimeOut* must have one of the values

`block` The call will block until a connection is established

`poll` This throws error 702 if immediate connection is not possible. By default this error simply fails but a user defined error handler can override this.

Integer The call will block for *Integer* seconds after which if no connection is established error 702 is thrown.

`tcp_listen(+`*Socket*`)`

In the case of the server in a connection oriented protocol after opening, and binding a socket it must specify a backlog for incoming connections from clients. The backlog size is 5. It corresponds to the C fragment:

```
{
        int socket=Socket;

        listen(socket, 5);
}
```

`tcp_socket(+`*Type*`, +`*Protocol*`, -`*Socket*`)`

This predicate is used to open a TCP socket. The variable *Type* must be one of

`sock_stream`
: For use in connected (stream) communication

`sock_dgram`
: For use in connectionless (datagram) communication

`sock_raw` For use in raw communication

The variable *Protocol* must be one of

`ipproto_ip`
: Suitable for any *Type*

`ipproto_udp`
: Suitable for *Type*=`sock_dgram`

`ipproto_tcp`
: Suitable for *Type*=`sock_stream`

`ipproto_icmp`
: Suitable for *Type*=`sock_raw`

`ipproto_raw`
: Suitable for *Type*=`sock_raw`

The variable *Socket* is bound to an integer which can be used in subsequent TCP operations.

`tcp_setsockopt(+`*Socket*`, +`*Optname*`, +`*Value*`)`

This predicate allows the setting of various options on sockets. The supported values of *Optname* are the following

`so_debug` turn on debugging info recording

`so_reuseaddr`
: allow local address reuse

`so_keepalive`
: keep connections alive

`so_dontroute`
: just use interface addresses

`so_broadcast`
: permit sending of broadcast msgs

`so_oobinline`
: leave received OOB data in line

`so_sndbuf`
: send buffer size

`so_rcvbuf`
: receive buffer size

`so_sndtimeo`
: send timeout

`so_rcvtimeo`
: receive timeout

The call is an interface to the UNIX system call `setsockopt(2)` and is equivalent to the following C fragment

```
{
        int val = Val, opt = Opt, sock = Sock;
        setsockopt(sock,SOL_SOCKET,opt,(char *)&val,sizeof(val));
}
```

`tcp_getsockopt(+`*Socket*`, +`*Optname*`, +`*Value*`)`

This predicate allows the getting of various options on sockets. The supported values of *Optname* are the following

`so_debug` turn on debugging info recording

`so_reuseaddr`
: allow local address reuse

`so_keepalive`
: keep connections alive

`so_broadcast`
: permit sending of broadcast msgs

`so_oobinline`
: leave received OOB data in line

`so_sndbuf`
: send buffer size

`so_rcvbuf`
: receive buffer size

`so_sndtimeo`
: send timeout

`so_rcvtimeo`
: receive timeout

`so_error`
: get error status and clear

`so_type`
: get socket type

The call is an interface to the UNIX system call `getsockopt(2)` and is equivalent to the following C fragment

```
{
        int val = Val, opt = Opt, sock = Sock;
        getsockopt(sock,SOL_SOCKET,opt,(char *)&val,sizeof(val));
}
```

3.6 Simple TCP Examples

3.6.1 Connected Sockets

The program below is an example of using the TCP connection protocol. The full source code can be found in the file '`$ICP_INSTALLDIR/examples/tcp/simple_connected.pl`'. A more sophisticated example is in file '`$ICP_INSTALLDIR/examples/tcp/connected.pl`'.

To run the example, two ICP processes should be started and the command

```
| ?- server.
```

run on one process while

```
| ?- client(Port, Address).
```

is run on the other. Input on the client is echoed by the server.

```
/********************** connected server ************************/
server :-
    tcp_server(0, Socket),
    tcp_getsockaddr(Socket, Port, Address),
    write('use '), write(client(Port, Address)), write('.\n'),
    server_loop(ok, Socket).

server_loop(stop, Socket) :- !, tcp_close(Socket).
server_loop(_, Socket) :-
    tcp_accept(Socket, NewSocket),
    process_request(ok, NewSocket, T),
    server_loop(T, Socket).

process_request(stop, Socket, stop) :- !, tcp_close(Socket).
process_request(end, Socket, end) :- !, tcp_close(Socket).
process_request(_, Socket, Final) :-
    tcp_recv(Socket, T),
    writenl(tcp_recv(Socket, T)),
    process_request(T, Socket, Final).
process_request(_, Socket, end) :- !, tcp_close(Socket).

/********************** connected client ************************/
client(Port, Address) :-
    tcp_client(Port, Address, Socket),
    client_loop(ok, Socket).

client_loop(stop, Socket) :- !, tcp_close(Socket).
client_loop(end, Socket) :- !, tcp_close(Socket).
client_loop(_, Socket) :-
    write('enter term, "end" ends this channel, "stop" stops server> '),
    flush,
    read(T),
    tcp_send(Socket, T),
    client_loop(T, Socket).
```

3.6.2 Connectionless Sockets

This example can be found in the file '`$ICP_INSTALLDIR/examples/tcp/connectionless.pl`'.

```
/************************ connectionless server ****************************/
```

```
server :-
    tcp_connectionless(0, Socket),
    tcp_getsockaddr(Socket, Port, Address),
    writeseqnl(['server Port:', Port, '  Address:', Address]),
    writeseqnl(['Server started.',
        'You can start a client with the following command: ','\n\t',
        client(Port, Address),'.']),
    server_loop(ok, Socket).

server_loop(stop, Socket) :- !, tcp_close(Socket).
server_loop(_, Socket) :-
    tcp_recvfrom(Socket, Data, Port, Address),
    writeseqnl(['got>', Data, '<', 'from Port:', Port,
                                '  Address:', Address]),
    server_loop(Data, Socket).

/************************ connectionless client ****************************/
client(Port, Address) :-
    tcp_connectionless(0, Socket),
    client_loop(ok, Socket, Port, Address).

client_loop(stop, Socket, _, _) :- !, tcp_close(Socket).
client_loop(end, Socket, _, _) :- !, tcp_close(Socket).
client_loop(_, Socket, Port, Address) :-
    write('enter term, "end" ends this channel, "stop" stops server> '),
    flush,
    read(Data),
    tcp_sendto(Socket, Data, Port, Address),
    client_loop(Data, Socket, Port, Address).
```

3.6.3 Broadcast Sockets

This example can be found in the file '`$ICP_INSTALLDIR/examples/tcp/broadcast.pl`'.

4 Mailbox-based communication model

TCP works well for applications which naturally map onto the client-server model. There is an asymmetry between the clients and the server. However, in the simple case of two processes having a conversation, it seems to be an unnecessary overhead and confusing complication to force one process to be the server and somehow provide a service to the client, when it is much more natural to think of them as two symmetrical processes.

In TCP, the instrument of communication is a socket. A socket is one end of a communication channel. To set up a channel between a server and a client, two sockets are created (one by the server and the other by the client) and are then connected to each other. This is analogous to a telephone conversation where two telephones are required to set up a connection. As with telephones, TCP communication is strictly one-to-one. There are no facilities to set up one-to-many or many-to-one conversations.

In TCP, every communication channel set up between a server and a client is a separate network connection. The system overheads of creating sockets and setting up connections over the network become significant if the conversations are short and there are many such conversations. For example, if ten processes on one machine are talking to ten processes on another machine, there will be ten network connections between the two machines.

These disadvantages can be overcome by providing a higher level abstraction that is more powerful and simpler to use. Mailboxes provide such an abstraction.

This section documents the Mailbox primitives. These primitives are available to both the Prolog and Parlog sub-systems of ICP. The Mailbox interface is not part of the core ICP system and must be loaded by a command of the form

```
?- ensure_loaded(mailbox).
```

The Prolog and Parlog interfaces are distinct libraries and so must be loaded separately.

Note that only the 'blocking' form of the mailbox primitives are currently implemented.

4.1 Introduction to Mailboxes

In the mailbox model, the instrument of communication is a mailbox. A mailbox is simply a repository for messages. Mailboxes may be created freely by any process. Messages can be sent to and removed from a mailbox. Instead of having to create two sockets to communicate, we now need to create only one mailbox. For two processes to communicate, the sender places a message in the mailbox, and the receiver removes it from the same mailbox. A mailbox can store multiple messages. Messages are kept in arrival order so that it naturally simulates stream communication.

4.1.1 Message Peeking

When a message is read from a mailbox, there is an option to not remove it. The message can remain in the mailbox. This is so that a receiving process can determine whether it should be left for another process to handle. This can occur because multiple processes may be reading from the same mailbox. When a message is being checked in this way, further reads from this mailbox must wait until a decision has been made regarding the message. The options are to commit the read and remove the message, or to discard the read and leave it in the mailbox.

4.1.2 Two-way Communication

Messages in a mailbox may be read by any process, including the process that sent it. This makes it difficult (though not impossible) to implement two-way communication using only one mailbox. Mailboxes are best used when only one receiver reads messages from it. Since there is very little cost associated with creating new mailboxes, two-way communication should be carried out using two mailboxes.

4.1.3 Multi-way Communication

A link may be established between two mailboxes. When a message is sent to the first mailbox, the message is automatically forwarded to the second mailbox. Note that the link is uni-directional only - messages sent directly to the second mailbox will not by forwarded to the first. The second mailbox remains an ordinary mailbox while the first becomes a ‘linked’ mailbox. Linked mailboxes do not store any messages, only addresses of links. A mailbox may be linked to many mailboxes. In this case, messages sent to the linked mailbox will cause a copy of the message to be forwarded to each and every link. This is how we can configure one-to-many communication. Note that there is a distinction between the case where multiple receivers each receive copies of all messages, and

the case where only one of the multiple receivers receive each message. In the former case, we can use linked mailboxes but in the latter case, one mailbox is sufficient.

Many-to-one communication can also be configured by linking multiple mailboxes to the same mailbox. Indeed, any arbitrary communication topology may be built up using links.

If a mailbox is linked to another mailboxes while some messages are stored in it (waiting to be read), the messages will be kept until the mailbox is unlinked or closed. If is unlinked, the messages will be accessible again.

4.2 Mailbox Server

All the above pre-supposes that a mailbox can be uniquely identified in a network and can be accessed by all machines on the network. This can be guaranteed by ensuring that when a mailbox is created, the identifier returned is a combination of the process number and the number of the mailbox in that process. The process number is allocated by an external mailbox server program. Note that the process number is a generalisation of the machine number since a machine may run many processes.

When a process requires the use of mailboxes, it first registers with the mailbox server which allocates a process number to it. The mailbox server replies by sending the process numbers and physical network addresses of all the other registered processes. It also notifies existing registered processes that there is a new process. By keeping all processes up-to-date with this information, processes may communicate with each other directly without contacting the mailbox server again. The mailbox server does not become a bottleneck in the system and in fact mailbox communication can continue even if the server program is terminated. Of course in this case, any further changes to the registered process list will not be reflected.

Using a mailbox server, a set of processes can cooperate and communicate with each other by registering with this server. The set of processes is called a *mailbox group*. A completely different mailbox group can exist if there is another mailbox server program running. In general, there is no limit to the number of mailbox groups. Each group is a completely independent system.

4.2.1 Registering Mailbox Names

In the model presented above, we can see how it would be possible to communicate if we know the mailbox identifier. However the bootstrapping problem of finding out the relevant identifier still remains.

If a process is willing to accept incoming communication, it should create a mailbox and register a name along with the mailbox identifier. This registration is done by sending a message to the mailbox server program (the mailbox server has a pre-defined address). Thereafter, when another process wishes to talk to the named process, it can query the mailbox server program for the mailbox identifier. This is very similar to telephone directories. The correspondence between names and telephone numbers is stored in a telephone directory. If you do not know the number, you look up the name. The mailbox server program provides this necessary directory lookup service.

4.3 Restricting Access

Restricted access can be enabled for the mailbox server and individual mailboxes. This is implemented in two layers.

The first layer is a user level restriction. At mailbox creation time, read and write permissions can be defined for the user, group and others. The possible values for permissions are `read_write`, `read`, `write` and `none`.

The second layer is a process level restriction. This filters the processes that pass the first layer. At mailbox creation time, a read password and/or a write password can be defined and every process wishing to access the mailbox must supply the correct password. The passwords are actually integers.

These restrictions can be used for the mailbox server as well, restricting the access to the services supplied.

4.4 Starting Mailboxes

The mailbox server is a standalone program written in C. To use mailboxes, there must first be a mailbox server running on the network. A mailbox server with the mailbox group name of `mbx1` is started in the background from the Unix command line like this :

```
mbx_server mbx1 &
```

Note that this is run from Unix, not from within Prolog. A confirmation message should appear as follows :

```
Server [mbx1] started on machine HOSTNAME
```

where HOSTNAME is the name of the machine on which the mailbox server is running. The program runs until it is explicitly killed using CTRL-C.

The identifier `mbx1` is associated with a particular port number in the file /etc/services. If you get the message

```
service: mbx1 not found
```

then /etc/services has not yet been modified. Please read the installation procedures on how to do this. There are four pre-defined mailbox group names in /etc/services, they are `mbx1`, `mbx2`, `mbx3` and `icprolog`. You may define more if you wish by adding lines to the /etc/services file.

When starting the server, if you get the message

```
error: server is already running or socket is in use
```

this means that the server is already running for that mailbox group. If however, you do not wish to participate in an existing mailbox group, then you should start another server using the identifiers `mbx2` or `mbx3` instead.

4.5 Mailbox Primitives

The Mailbox primitives described below are available to both Prolog and Parlog components of the ICP system.

In the short forms of the primitives, the dafault values for the arguments are user `read_write`, group `write` and others `write` permissions, `0` read password and `0` write password and they supply `0` as password.

4.5.1 Opening, Modifying and Closing Mailboxes

`mbx_init(+`*Address*`, +`*MailboxGroup*`)`

Connects to server *MailboxGroup* on machine *Address*. This must be the first mailbox primitive to be called since this initialises the connection. *MailboxGroup* is normally one of `mbx1`, `mbx2` or `mbx3`, but it may be any string as long as it is defined in the /etc/services file.

`mbx_create(-`*Mbx*`)`
`mbx_create(-`*Mbx*`, +`*User*`, +`*Group*`, +`*Others*`, +`*RPass*`, +`*WPass*`)`

This creates a mailbox, returning its global address as an identifier *Mbx*. For efficiency reasons, it is more natural (although not necessary) to create a mailbox on the machine that the processes which are going to receive from it are running.

`mbx_close(+`*Mbx*`)`
`mbx_close(+`*Mbx*`, +`*TimeOut*`, +`*RPass*`)`

This primitive closes the mailbox specified, breaking all the links from or to it.

`mbx_link(+`*MbxI*`, +`*MbxO*`)`
`mbx_link(+`*MbxI*`, +`*RPass*`, +`*MbxO*`, +`*WPass*`, +`*TimeOut*`)`

This links one mailbox (*MbxI*) to another (*MbxO*), allowing one to one, many to one, one to many or many to many communication. Linking a mailbox to itself gives an error.

`mbx_unlink(+`*MbxI*`, +`*MbxO*`)`
`mbx_unlink(+`*MbxI*`, +`*RPass*`, +`*MbxO*`, +`*WPass*`, +`*TimeOut*`)`

This breaks the link between *MbxI* and *MbxO*.

`mbx_getlinks(+`*Mbx*`, -`*Links*`)`
`mbx_getlinks(+`*Mbx*`, +`*TimeOut*`, +`*RPass* `, -`*Links*`)`

Not implemented yet. This returns the links of the specified mailbox. *Links* is a list with two elements, the first is a list with the addresses of mailboxes to which `Mbx` is linked to, and the second element is a list with the addresses of mailboxes linked to *Mbx*.

4.5.2 Reading and Writing from Mailboxes

`mbx_send(+`*Mbx*`, ?`*Term*`)`
`mbx_send(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, ?`*Term*`, +`*WPass*`)`

This sends a term to the mailbox specified by the global address *Mbx*. Sending is asynchronous (and therefore buffering is provided).

`mbx_recv(+`*Mbx*`, -`*Term*`)`
`mbx_recv(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, -`*Term*`, +`*RPwd?*`)`

This reads a term from the mailbox specified by the global address *Mbx*. *Flag* specifies whether the message to be received is out of band data or normal data. *TimeOut* specifies whether call is in `block`, `poll` or timeout (*TimeOut* is a number in seconds) mode. `block` means that the call will suspend until a message is available. `poll` means that if there is no message available, the call will fail with TIMEOUT error. *TimeOut* means that the call will suspend for a message for at most the time specified and then, if there is no message available, it will fail. *RPwd* is an integer password for reading the mailbox.

`mbx_look(+`*Mbx*`, -`*Term*`)`
`mbx_look(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, -`*Term*`, +`*RPass*`)`

This reads a term from the mailbox *Mbx*. It is a blocking primitive. A `mbx_look` call does not remove the message read. The mailbox is locked after such a call, so any other `mbx_read` or `mbx_look` calls suspend until the mailbox is unlocked. The reading process is supposed to test the message read and, depending on whether it is suitable or not, it should call `mbx_commit` or `mbx_discard`, both of which unlock the mailbox. `mbx_commit` or `mbx_discard` (whichever is chosen) should be run sequentially after `mbx_look`, otherwise a SEQUENCE error will occur.

`mbx_commit(+`*Mbx*`)`
`mbx_commit(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, +`*RPass*`)`

This unlocks the mailbox *Mbx* removing the message read, so the next reader will read a different message.

`mbx_discard(+`*Mbx*`)`
`mbx_discard(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, +`*RPass*`)`

This unlocks the mailbox *Mbx* without removing the message read, so the next reader will read the same message.

`mbx_check(+`*Mbx*`)`
`mbx_check(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, +`*RPass*`)`

This primitive checks whether there is a message available in the mailbox specified. It is a non-blocking primitive.

`mbx_clear(+`*Mbx*`)`
`mbx_clear(+`*Mbx*`, +`*Flag*`, +`*TimeOut*`, +`*Pass*`)`

This removes all the normal data or out of band data kept in the specified mailbox.

4.5.3 Miscellanious Mailbox Predicates

`mbx_bind(+`*Mbx*`, +`*Atom*`)`
`mbx_bind(+`*Mbx*`, +`*TimeOut*`, +`*Atom*`, +`*RPass*`)`
: This binds a global mailbox address to a global service in the mailbox server, making the address accessible to the whole system. There can be different services (*Atom*) with the same address (*Mbx*), but not one service with different addresses.

`mbx_getid(+`*Atom*`, -`*Mbx*`)`
`mbx_getid(+`*Atom*`, +`*TimeOut*`, -`*Mbx*`, +`*Pass*`)`
: This returns the global mailbox address associated with the global service specified. If the service is not recorded in the internal data base, the primitive fails.

`mbx_getname(+`*Mbx*`, -`*Atom*`)`
`mbx_getname(+`*Mbx*`, +`*TimeOut*`, -`*Atom*`, +`*Pass*`)`
: This returns the global service associated with the global mailbox address specified. If the address is not recorded in the internal data base, the primitive fails.

`mbx_initdb(-`*Ptr*`)`
`mbx_initdb(-`*Ptr*`, +`*TimeOut*`)`
: This initializes the internal data base reading pointer, allowing sequential access to it using the `mbx_getdb/4` primitive. Returns an identifier for the initialize pointer.

`mbx_getdb(+`*Ptr*`, -`*Atom*`, -`*Mbx*`)`
`mbx_getdb(+`*Ptr*`, +`*TimeOut*`, -`*Atom*`, -`*Mbx*`, +`*RPass*`)`
: This returns the global service and the global mailbox address associated with it, reading sequentially from the internal data base. The next call to this primitive will return the next data base entry. *Ptr* is the identifier for the internal data base pointer.

`mbx_closedb(+`*Ptr*`)`
`mbx_closedb(+`*Ptr*`, +`*TimeOut*`)`
: It closes the internal data base reading pointer created by `mbx_initdb/1`.

The last two primitives always succeed, except when the arguments are incorrect.

5 Foreign Language Interface

ICP offers an interface to functions written in the programming language 'C' — Brian W. Kerninghan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, inc. Currently the calling of 'C' functions from prolog is supported but 'C' can not call prolog. In later versions this support will be provided. Dynamic loading is achieved through the unix '`ld`' command.

The arguments of predicates corresponding to 'C' functions are restricted to non complex terms - integers, floats or atoms. The type of each argument must be specified together with whether it is an input or output argument. Input arguments must be ground when called, while output arguments may also be variables. The interface provides the transformation to and from the internal Prolog representation of the primitive types and the corresponding 'C' representation.

5.1 Foreign Language Predicates

`load_foreign_files(+`*FileList*`)`

This is used to load foreign functions into the running prolog system. *FileList* is a list of object files (compiled 'C' files). The "`tt .o`" extension may be ommited. A single atom (file name) is also allowed as a value of *FileList*. The files are searched for in the current search path.

`load_foreign_files(+`*FileList*`, +`*LinkerArgs*`)`

Same as above with the additional parameter *LinkerArgs* specifying a list of arguments to be passed to the unix '`ld`' command. This is typically used when additional system libraries are required.

The user defined predicates `foreign_file/2` and `foreign/3` are used to specify the prolog predicates and corresponding 'C' functions.

`foreign_file(-`*File*`, -`*Functions*`)`

File is an object (compiled 'C') file while *Functions* is a list of the 'C' function names defined in this file which are to be interfaced.

`foreign(+`*Cfunction*`, +`*Language*`, +`*PredicateSpecification*`)`

Cfunction is the name of a 'C' function and *Language* is the atom `c` (In future languages other than 'C' may be supported). *PredicateSpecification* describes the form of the prolog predicate to be used to access the foreign function. This is a term of the form PrologName(*ArgSpec1*, ..., *ArgSpecN*)
Each argument is of one of the following forms

- `+`*integer* The Prolog argument is an integer. The corresponding 'C' argument type is `long int`.

- -*integer* The Prolog argument is an integer or a variable. The corresponding 'C' argument type is `long int *`.
- `[`-*integer*`]` The Prolog argument is an integer or a variable. The corresponding 'C' function type is `long int *`.
- +*float* The Prolog argument is a floating point number. The corresponding 'C' argument type is `double`.
- -*float* The Prolog argument is a floating point number or a variable. The corresponding 'C' argument type is `double *`.
- `[`-*float*`]` The Prolog argument is a floating point number or a variable. The corresponding 'C' function type is `long int *`.
- +*atom* The Prolog argument is an atom. The corresponding 'C' argument type is `char *`.
- -*atom* The Prolog argument is an atom or a variable. The corresponding 'C' argument type is `char **`.
- `[`-*atom*`]` The Prolog argument is an atom or a variable. The corresponding 'C' function type is `char *`.
- +*string* Synonym for +*atom*.
- -*string* Synonym for -*atom*.
- `[`-*string*`]` Synonym for `[`-*atom*`]`.

The 'C' function must include the file '`ICprolog.h`'. There should be no more than one argument corresponding to a 'C' function return type. If there is no such argument, the return type of the 'C' function should be specified as `bool`, and it should return one of the following values

`FAIL` The prolog call will fail.

`SUCCEED` The prolog call will succeed.

`SUSPEND` If a primitive returns `SUSPEND`, then that thread is taken out of the run queue. This means that another thread will have to explicitly 'resume' See Section 2.15 [Threads], page 50, it before it can run again.

`REQUEUE` `REQUEUE` means the primitive should be tried again later. This can result in a busy wait if nothing happens in between to change the status.

5.1.1 Compilation and Example

When compiling the `-I` option should be given to the 'C' compiler to specify the directory where the '`ICprolog.h`' file is located. For example

```
cc -w -O -I$ICP_INSTALLDIR/include -c myfuncs.c
```

With the following code for '`myfuncs.c`'

```
#include "ICprolog.h"

char *malloc();

/* foreign(twice, c, ptwice(+float, [-float])). */
double twice(a)
double a;
{
        return(2 * a);
}

/* foreign(concat, c, pconcat(+atom, +atom, -atom)). */
bool concat(a, b, c)
char *a, *b, **c;
{
        int len;
        static int last_len = 0;
        static char *ret = NULL;
        char *pt;

        len = strlen(a) + strlen(b) + 1;
        if (!ret) {
                last_len = len;
                ret = malloc(len);
        } else if (len > last_len) {
                free(ret);
                ret = malloc(len);
                last_len = len;
        }
        pt = ret;
        while (*pt = *a++)
                pt++;
        while (*pt = *b++)
                pt++;
        *c = ret;
        return(SUCCESS);
}

/* foreign(foo, c, pfoo(+integer,-integer,+atom,-atom,+float,-float)). */
bool foo(int1, int2, atom1, atom2, float1, float2)
int int1, *int2;
char *atom1, **atom2;
double float1, *float2;
```

```
{
        *int2 = int1;
        *atom2 = atom1;
        *float2 = float1;
        return(SUCCESS);
}
```

and the following prolog declarations

```
foreign_file('test.o', [twice, concat, foo]).

foreign(twice, c, ptwice(+float, [-float])).
foreign(concat, c, pconcat(+atom, +atom, -atom)).
foreign(foo, c, pfoo(+integer, -integer, +atom, -atom, +float, -float)).
```

will give the results:

```
| ?- [test], listing([foreign_file, foreign]).
foreign_file('test.o',[twice,concat,foo]).

foreign(twice,c,ptwice(+ float,[- float])).
foreign(concat,c,pconcat(+ atom,+ atom,- atom)).
foreign(foo,c,pfoo(+ integer,- integer,+ atom,- atom,+ float,- float)).

yes
| ?- load_foreign_files('test.o').
{loading foreign from ./test_interface.o ./test.o }

yes
| ?- ptwice(1.1, T).

T = 2.2

| ?- pconcat(ab, cd, E).

E = abcd

| ?- pfoo(4, I, hello, A, 1.3, F).

I = 4
A = hello
F = 1.3
```

6 Prolog Tracer User Guide

When programs do not work as expected, the next step is to use a Prolog tracer to trace through the execution step by step. Our tracer uses an extended set of the four port model described by Clocksin and Melish *Programming in Prolog*, Springer Verlag.

The tracer knows about the following eight ports: call, exit, redo, fail, unify, try_match, succeed_match, and fail_match. On invoking the tracer the first five of these are set on leash. Leashed ports are always echoed, and skipping and unleashing are allowed at call and redo ports only. At leashed ports the '?' prompt awaits for the user to take action by typing one of `RET s u n a e l p d !` The effect is as follows:

`RET` creeping takes place

`s`	the current goal is skipped
`u`	the current goal is unleashed
`n`	the trace continues by
`a`	tracing is aborted
`e`	the predicate set_leashes is called
`l`	the predicate set_lechoes is called
`p`	information is displayed indicating the ports that are currently leashed/echoed
`d`	the ICP debugger is entered
`!`	the user is allowed to solve a prolog query. On the user selecting this option the tracer types ! and awaits for a prolog query to be typed at the terminal. Once the query has been solved, control returns to the tracer.

`trace` Turns the tracer on. All top level queries will be traced.

`notrace` Turns the tracer off.

`set_echoes`

Gives the user the opportunity to turn echoes on or off on any of the tracer's ports.

set_echoes types the message '...ports to be leashed?' and awaits for the user to type a term of the form `<Ports>`:

`<Ports>` ::= all | none | <PortSeq>

`<PortSeq>`

::= <Port> | -<Port> | <Port>,<Ports> | -<Port>,<Ports>

`<Port>` ::= call | exit | redo | fail | try_match | succeed_match | fail_match

If the term is 'all' then the echo is set to be on all the tracer's ports.

If the term is 'none' then the echo and leash is set to be off on all tracer's ports.

If the term has a subterm of the form <Port> then the echo is set to be on on the corresponding port.

If the term has a subterm of the form -<Port> then the echo and leash are set to be off on the corresponding port.

The port names call, exit, redo, fail, try_match, succeed_match, and fail_match can be respectively abbreviated to c, e, r, f, tm, sm, and fm.

`set_leashes`

Gives the user the opportunity to turn leashes on or off on any of the tracer's ports.

set_echoes types the message '...ports to be leashed?' and awaits for the user to type a term of the form <Ports> (see set_echoes).

If the term is 'all' then the echo and leash is set to be on all the tracer's ports.

If the term is 'none' then the leash is set to be off on all tracer's ports.

If the term has a subterm of the form <Port> then the echo and leash are set to be on on the corresponding port.

If the term has a subterm of the form -<Port> then the leash is set to be off on the corresponding port.

`??` +*Goal* *Goal* is traced. This operator can be used to trace a single goal when the tracer is off. Only dynamic code can be traced and if during a trace a query for a static predicate is encountered, the query is solved by a standard call to prolog.

Appendix A Operators

Precedence	Type	Name
1200	yfy	(.)
1200	yf	(.)
1200	xfx	(-->) (:-)
1200	fx	(:-) (?-)
1150	fx	(multifile) (mode) (public) (meta_predicate) (dynamic)
1100	xfy	(;)
1050	xfy	(->)
1000	xfy	(,)
900	fy	(nospy) (\+) (not) (spy)
900	fx	(??)
700	xfx	(<) (=) (>) (=<) (==) (=:=) (@<) (@>) (@=<) (@>=)
		(\=) (\==) (=..) (>=) (=\=) (is)
600	xfy	(:)
500	yfx	(\/) (++) (--) (/\) (-) (+)
500	fx	(\) (-) (+)
400	yfx	(<<) (**) (//) (*) (/) (>>)
300	xfx	(mod)
200	xfy	($\hat{}$)
100	fx	(ls) (cd) (vi)

Appendix B Error Codes

```
100 unbound variable in arithmetic expression
101 unbound variable in relational expression
102 unbound stream identifier
103 unbound filemode

200 non-integer in arithmetic expression
201 invalid clause
202 non-number in relational expression
203 non-atom filename
204 non-atom in operator declaration
205 non-variable stream identifier
206 invalid data for ram file
207 non-integer argument in throw/1
208 non-variable in 2nd argument
209 non-variable argument
210 invalid form of use
211 unterminated list
212 invalid list of ASCII codes
213 non-integer argument
214 attempt to assert/retract a static predicate
215 attempt to assert/retract a system predicate
216 invalid clause
217 invalid goal
218 non-number in arithmetic expression
219 invalid arithmetic expression

300 cannot open file
301 invalid stream
302 cannot close stream
303 incorrect or unknown file type
304 error occurred while reading object file
305 error occurred while writing object file
306 invalid ram file
307 invalid port
308 port has pending read
309 port has pending write
310 port is being read by another thread
311 port is being written to by another thread
312 port is owned by another thread

400 file mode must be read, write or append
401 operator priority must be in the range 0..1200
402 operator type must be fx, fy, xf, yf, xfx, xfy, yfx or yfy
403 predicate not defined
404 undefined property
405 undefined predicate
408 system deadlock
```

```
409 invalid return code from service function
410 parlog thread exists already
411 invalid thread ID
412 cannot switch to parlog from this thread

500 compiler error
501 invalid instruction found during code generation
502 multiply-defined label found during code generation
503 term nested too deep
504 out of stack space
505 out of heap space
506 invalid ICP instruction
507 invalid tag type found
508 too many variables in term
509 term too large in property
510 term too large for pipe

600 division by zero in arithmetic expression
601 cannot open any more file streams
602 cannot open any more memory streams
603 no space for RAM file
604 code segment too large
605 symbol space full during code generation
606 out of space during code generation
607 out of code space
608 out of space in dictionary maintenance
609 out of space for constants
610 no space for new thread
611 no space for property
612 out of space in term reader
613 out of system heap space
614 illegal instruction

700 Bad sequence of TCP instructions
701 Socket closed
702 Timeout (TCP)
703 Invalid socket descriptor
710 socket error (TCP)
711 setsockopt error (TCP)
712 bind error (TCP)
713 listen error (TCP)
714 accept error (TCP)
715 connect error (TCP)
716 ioctl error (TCP)
717 send error (TCP)
718 recv error (TCP)
719 sendto error (TCP)
720 recvfrom error (TCP)
721 close error (TCP)
722 getsockname error (TCP)
```

```
723 gethostname error (TCP)
724 gethostbyname error (TCP)
725 getpeername error (TCP)
726 gethostbyaddr error (TCP)
727 getservbyname error (TCP)
728 getservbyport error (TCP)
729 getsockopt error (TCP)
730 cannot open any more sockets
731 data too long for buffer

800 Maibox initialisation
801 Timeout (mailbox)
802 Incorrect mailbox identifier
803 Mailbox permission denied
804 Invalid operation on linked mailbox
805 The mailbox is locked
806 The mailbox is empty
807 Attempt to commit or discard an unlocked mailbox
808 The machine on which the mailbox is located is shut down
809 An error has happened in the low level communications
810 Attempt to bind a mailbox to a service which already exists
811 The specified service does not exist in the internal database
812 The specified mailbox does not exist in the internal database
813 Unspecified Mailbox error
814 Unification failure error (mailbox)
```

Appendix C Low-level Primitives

```
noref                              % list undefined preds
debugicp                           % debug emulator
halt                               % exit emulator
'add%f'(N,M,Sum)                   % add two integers
'sub%f'(N,M,Difference)            % subtract two integers
'mul%f'(N,M,Product)               % multiply two integers
'div%f'(N,M,Quotient)              % divide two integers
'mod%f'(N,M,Remainder)             % modulus
'cmp%f'(Order,term1,Term2)         % compare two terms
'int%f'(Int)                       % integer test
'atom%f'(Constant)                 % atomic test
'var%f'(Var)                       % var test
'list%f'(List)                     % list test
'tpl%f'(Tuple)                     % tuple test
'arg%f'(N,Term,Element)            % extract from tuple
'univ%f'(Tuple,List)               % univ
'name%f'(Constant,List)            % print-names of terms
'o_stream%f'(Filename,Mode,Id)     % initialise stream I/O
'c_stream%f'(Id)                   % finished with stream
'o_mem%f'(Initialiser,Mode,Id)     % initialise memory file
'c_mem%f'(Id)                      % finished with memory file
'set_in%f'(Type,Id)                % set current input stream
'set_out%f'(Type,Id)               % set current output stream
'increment%f'(N,OneMore)           % increment by 1
'put_q_atom%f'(Atom)               % write out a quoted atom
'get0%f'(Char)                     % input a character
'put%f'(Char)                      % output a character
'unget%f'(Char)                    % put back a character
'flush%f'(StreamId)                % flush output stream
'addr%f'(Var,Address)              % numerical address of var
'deepcut%f'(Cut)                   % deep cut
'back%f'(ChoicePt)                 % pick up cut label
'meta%f'(Goal)                     % meta-call
'eq%f'(Number,Number)              % equality test
'ne%f'(Number,Number)              % inequality test
'lt%f'(Number,Number)              % less than test
'le%f'(Number,Number)              % less than or equal test
'gt%f'(Number,Number)              % greater than test
'ge%f'(Number,Number)              % greater than or equal test
'syntax_error%f'(Errorcode)        % Syntax Error Handler
'assert%f'(Index,Mode)             % link clause of dynamic predicate
'retract%f'(Name,Arity,Index)      % retract dynamic clause
'dec%f'([N])                       % decrement N by 1 in [N]
'inc%f'([N])                       % increment N by 1 in [N]
'find_clause%f'(Name,Arity,ChainName,ChainArity,Index)
                                   % find specific clause in dynamic chain
'time%f'(Time)                     % get CPU time
```

```
'arity%f'(Term,Arity)           % get arity of tuple
'funct%f'(Term,Functor)         % get functor of tuple
'interm%f'(Term,Priority,Vars)  % read a term and produce var list
'ntpl%f'(Length,Tuple)          % create a new tuple of unbounds
'not%f'(N,Complement)           % binary complement
'lshift%f'(N,Places,Result)     % left shift function
'rshift%f'(N,Place,Result)      % right shift function
'and%f'(N,M,Anded)              % and function
'or%f'(N,M,Or)                  % or function
'op_prefix%f'(OpPrior,Right,Name)     % define a new prefix operator
'op_postfix%f'(Left,OpPrior,Name)     % define a new postfix operator
'op_infix%f'(Left,OpPrior,Right,Name) % define a new infix operator
'op_look%f'(Name,Pre,PreRight,InLeft,In,InRight,Post,PostLeft)
                                % read specified operator details
'op_get%f'(Name,IndexIn,Pre,PreRight,InLeft,In,InRight,
          Post,PostLeft,IndexOut).
                                % read unspecified operator details
'put_str%f'(List,Quote)         % write a quoted atom (list of ASCII)
'put_atom%f'(Atom)              % write out an atom
'put_number%f'(Number)          % write out a number
'atom_type%f'(Constant,Type)    % is atom alphanumeric or symbols ?
'defined%f'(Pred,Address)       % return address of predicate
'read_catch%f'(ChoicePt)        % read CATCH register
'set_catch%f'(ChoicePt)         % set CATCH register
'read_error%f'(Error)           % read and reset ERROR register
'catch%f'(B)                    % set CATCH to most recent chpt.
'throw%f'(Error)                % throw an error
'cg_init%f'(Type)               % initialise code generator
'cg_fixup%f'(Public)            % fixup generated code
'cg%f'(Instructions)            % generate code for one relation
'undef%f'(Name,Arity)           % get name and arity of undfnd relation
'c_ram%f'(Ramfile,Constant)     % create constant from memory file
'load%f'                        % load code block from current input
'varsin%f'(Term,Vars)           % Vars is list of variables in Term
'fix_tables%f'                  % fixup externals and entries tables
'save%f'(PredList,Saved,Unsaved)   % save code for pred to current output
'pred_look%f'(Name,Arity,Seg,Type) % is specified predicate defined ?
'pred_get%f'(Name,Arity,IndexIn,IndexOut,Seg,Type)
                                % Get unspecified predicate details
'abolish%f'(Name,Arity)         % Abolish predicate
'validate%f'(DefindLst,ErrType) % Gets a list of preds being re-loaded
'unload%f'                      % Free space for last loaded segment
'kill%f'(Name,Arity)            % Kill predicate
'write_term%f'(Term)            % write encoded term
'read_term%f'(Term)             % read encoded term
'tag%f'(Term,Tag)               % return tag as in integer
'grnd_funct%f'(Functor)         % test for ground functor
'tab%f'(Number)                 % print spaces
'concat%f'(First,Second,Join)   % concat two symbols
'prefix%f'(Longname,Prefix)     % symbol prefix test
```

```
'suffix%f'(Longname,Suffix)     % symbol suffix test
'to_ground%f'(Hllw,Grnd,Vr,VNm,Used) % create ground copy of term
'to_hollow%f'(Grnd,Hllw,VNames,Vars) % creates hollow copy of a term
'delete%f'(Name,Arity)          % free space for clauses of  Name/Arity
'cg_out%f'                      % write out result of code generator
'set_up_seg%f'                  % initialise segment from ram file
'tty%f'(Name,InStream,OutStream) % create titled window with I/O streams
'ram_pipe%f'(Out,In)            % create pseudo-pipe using ram files
'close_port%f'(Port)            % close one end of a pipe
'read_pipe%f'(Port,Term)        % read a term from a pipe
'write_pipe%f'(Port,Term)       % write a term to a pipe
'new_thread%f'(Thread)          % create a new thread
'kill_thread%f'(Thread)         % kill a thread
'fork%f'(Thread,InputStream)    % fork a thread
'resume%f'(Thread)              % resume a thread
'suspend%f'                     % suspend a thread
'rpc%f'(Thread,InputStream,OutputStream) % remote procedure call
'pipe%f'(In,Out)                % create a pipe using memory files
'thread%f'(Thread)              % get address of current thread
'set_prop%f'(Object, Property, Value) % setting a property
'get_prop%f'(Object, Property, Value) % retrieving a property
'del_prop%f'(Object, Property)  % deleting a property
'get_props%f'(Object, List)     % get property names of an object
'del_props%f'(Object)           % delete all properties of an object
'get_cons%f'(Property, List)    % get objects which have property
'del_cons%f'(Property)          % delete objects which have property
'float%f'(Num)                  % floating-point number test
'put_float%f'(Num)              % write out a floating-point
'number%f'(Num)                 % integer/float test
'fadd%f'(N,M,Sum)               % add two floating point numbers
'fsub%f'(N,M,Difference)        % subtract two floating point numbers
'fmul%f'(N,M,Product)           % multiply two floating point numbers
'fdiv%f'(N,M,Quotient)          % divide two floating point numbers
'int%f'(Float,Int)              % convert float to integer
'sin%f'(N,Result)               % sine function
'cos%f'(N,Result)               % cosine functions
'tan%f'(N,Result)               % tangent function
'asin%f'(N,Result)              % arcsin function
'acos%f'(N,Result)              % arcosine function
'atan%f'(N,Result)              % arctangent function
'atan2%f'(N,M,Result)           % arctangent (N/M) function
'sinh%f'(N,Result)              % hyperbolic sine function
'cosh%f'(N,Result)              % hyperbolic cosine function
'tanh%f'(N,Result)              % hyperbolic tangent function
'exp%f'(N,Result)               % natural anti-logarithm function
'log%f'(N,Result)               % natural logarithm function
'log10%f'(N,Result)             % logarithm (base 10) function
'sqrt%f'(N,Result)              % square root function
'pow%f'(N,M,Result)             % N raised to the power M
'abs%f'(N,Result)               % absolute value function
```

```
'sign%f'(N,Result)              % sign of N
'ceil%f'(N,Result)              % round up to next higher integer
'floor%f'(N,Result)             % round down to next lower integer
'pi%f'(Pi)                      % value of pi
'rand%f'(N)                     % random integer in range 0-32767
'deg2rad%f'(N,Result)           % degree to redian conversion
'rad2deg%f'(N,Result)           % radian to degree conversion
'interrupt%f'(Goal)             % retrieve interrupted goal
'unix%f'(Command)               % execute unix command
'out_of_date%f'(File)           % check if need to recompile
'look_pipe%f'(Port,Term)        % read from pipe without removing
'commit_read%f'(Port)           % commits the non-removing read
'look_pipe%f'(Port,Term)        % unlock a pipe
'init_parlog%f'(Th)             % start up a Parlog thread
'tty_get0%f'(Char)              % read character from terminal
'curr_input%f'(In)              % identify current input
'curr_output%f'(Out)            % identify current output
'release_port%f'(Port)          % relinquish ownership of port
'empty_pipe%f'(Pipe)            % test if pipe is empty
'is_iport%f'(Port)              % input port test
'is_oport%f'(Port)              % output port test
'runtime%f'(Flag, List)         % get CPU time (SICSTUS style)
'decrement%f'(N,OneLess)        % decrement by 1
'load_foreign%f'(File,List,Lib) % set up foreign C primitives
'realtime%f'(Number)            % real time (seconds since 1/1/170)
'errno%f'(Error,Msg)            % C errno
'getenv%f'(Name, Value)         % get environment variable
```

Predicate Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

User Defined Predicate Index

Concept Index

Short Contents

Table of Contents