

# KLIC User's Manual

---

March 1995

(Revised at June 1997)

This manual corresponds to KLIC version 3.002

Takashi Chikayama (Tokyo University)

Tetsuro Fujise (Mitsubishi Research Inst., Inc.)

Daigo Sekita (Mitsubishi Research Inst., Inc.)

---



# Table of Contents

<b>Terms and Conditions for Use of ICOT Free Software .....</b>	<b>1</b>
<b>1 Introduction .....</b>	<b>3</b>
1.1 Description of Predicates and Methods .....	3
1.1.1 Predicates and Methods .....	3
1.1.2 Messages .....	4
1.1.3 Argument Modes .....	4
1.2 Reporting Bugs and Sending Comments .....	4
<b>2 KL1 Language .....</b>	<b>5</b>
2.1 Basic Execution Mechanism .....	5
2.2 Predicates .....	6
2.3 Modules .....	6
2.4 Goals .....	7
2.5 Initial Goal .....	7
2.6 Generic Objects .....	7
2.6.1 Creating Generic Objects .....	8
2.6.2 Guard Methods of Generic Data Objects .....	8
2.6.3 Body Methods of Generic Data Objects .....	8
2.7 Priority Specification .....	8
2.8 Clause Preference .....	9
2.9 Shorthand Notation for Argument Pairs .....	9
2.9.1 Paired Arguments and their Expansion .....	10
2.9.2 Macros for Paired Arguments .....	11
2.9.3 Usage of Paired Arguments .....	11
2.10 Inserting C Language Code Inline .....	12
2.10.1 Inline Insertion at the Top of Files .....	12
2.10.2 Inline Insertion in the Guard .....	12
2.10.3 C-Level Representation of KL1 Terms .....	13
2.10.4 Examples .....	13
2.10.5 Some Hints on Using the Inline C Code Feature .....	14
<b>3 Builtin and Library Features .....</b>	<b>15</b>
3.1 Common Operations .....	15
3.1.1 Unification .....	15
3.1.2 Synchronization .....	15
3.1.3 Comparison and Hashing .....	15
3.1.4 Execution Status .....	16
3.1.5 Debugging .....	16
3.2 Atomic Data .....	17

3.2.1	Symbolic Atoms .....	17
3.2.1.1	Notation of Symbolic Atoms .....	17
3.2.1.2	Operations on Symbolic Atoms .....	18
3.2.2	Integer Atoms .....	19
3.2.2.1	Notation of Integers .....	19
3.2.2.2	Integer Arithmetics .....	19
3.2.2.3	Integer Comparison .....	20
3.2.3	Floating Point Numbers .....	20
3.2.3.1	Notation of Floating Point Numbers .....	21
3.2.3.2	Creating New Floating Point Numbers .....	21
3.2.3.3	Floating Point Arithmetics .....	21
3.2.3.4	Floating Point Comparison .....	22
3.3	Structured Data .....	23
3.3.1	Functor Structures .....	23
3.3.1.1	Notation of Functors .....	23
3.3.1.2	Operations on Functors .....	23
3.3.2	Lists .....	24
3.3.2.1	Notation of Lists .....	25
3.3.2.2	Manipulation of Message Streams .....	25
3.3.3	Vectors .....	27
3.3.3.1	Notation of Vectors .....	27
3.3.3.2	Creating New Vectors .....	27
3.3.3.3	Predicates on Vectors .....	27
3.3.4	Strings .....	28
3.3.4.1	Notation of Strings .....	28
3.3.4.2	Creating New Strings .....	29
3.3.4.3	Predicates on Strings .....	30
3.4	Handling Program Code as Data .....	31
3.4.1	Modules .....	31
3.4.2	Predicates .....	31
3.5	Unix Interface .....	32
3.5.1	Obtaining Unix Interface Stream .....	32
3.5.2	Opening Streams for Input and Output Operations .....	33
3.5.3	Using Sockets .....	34
3.5.4	Files and Directories .....	34
3.5.5	Handling Signal Interrupts .....	35
3.5.6	Miscellaneous Messages to the Unix Stream .....	36
3.5.7	Predicate Interface .....	36
3.6	Input and Output .....	37
3.6.1	Input and Output with C-like Interface .....	37
3.6.1.1	Common Messages with C-like Interface .....	37
3.6.1.2	Input Messages with C-like Interface .....	38
3.6.1.3	Output Messages with C-like Interface .....	38
3.6.2	Input and Output with Prolog-like Interface .....	39
3.6.2.1	Opening Prolog-like I/O Streams .....	39
3.6.2.2	Common Messages with Prolog-like Interface .....	39
3.6.2.3	Input Messages with Prolog-like Interface .....	40
3.6.2.4	Output Messages with Prolog-like Interface .....	40

3.6.2.5	Wrapped Terms.....	41
3.7	Controlling System Behavior.....	42
3.8	Timer.....	43
3.9	Random Number Generator.....	44
<b>4</b>	<b>Using KLIC.....</b>	<b>45</b>
4.1	Compiling Programs with KLIC.....	45
4.1.1	Command for Compilation.....	45
4.1.2	Compiler Options.....	45
4.1.3	How KLIC Compiler Works.....	47
4.2	Running Programs Compiled with KLIC.....	47
4.2.1	Runtime Switches for Programs Compiled with KLIC.....	48
4.3	Tracing Program Execution.....	48
4.3.1	Preparation for Traced Execution.....	48
4.3.2	Trace Ports.....	49
4.3.3	Format of Trace Display.....	49
4.3.4	Trace Controlling Commands.....	51
4.3.4.1	Controlling Tracing of the Traced Goal.....	51
4.3.4.2	Controlling Tracing of Newly Created Subgoals.....	52
4.3.4.3	Changing Default Trace of Predicates.....	52
4.3.5	Spying.....	53
4.3.6	Controlling Trace Ports.....	53
4.3.7	Display Control Commands.....	54
4.3.8	Dumping Goals.....	55
4.3.9	Miscellaneous Commands.....	55
4.3.10	Detecting Perpetual Suspensions.....	55
4.4	Installation.....	56
4.4.1	Configuration.....	56
4.4.2	Compiling the KLIC system.....	56
4.4.3	Testing the Compilation.....	56
4.4.4	Installing the Objects.....	57
4.4.5	Cleaning Up the Installation Directory.....	57
4.4.6	When Something Goes Wrong.....	57
4.5	Distributed Memory Parallel Implementation of KLIC.....	57
4.5.1	Installation of Distributed KLIC.....	57
4.5.2	Compiling Programs for Distributed KLIC.....	58
4.5.3	Running Programs of Distributed KLIC.....	58
4.5.3.1	Setting Up PVM.....	58
4.5.3.2	Runtime Options for Distributed KLIC.....	58
4.5.3.3	Known Bugs of Distributed KLIC.....	59
4.6	Shared-Memory Implementation of KLIC.....	59
4.6.1	Installation of Shared-Memory KLIC.....	59
4.6.2	Compiling Programs for Shared-Memory KLIC.....	60
4.6.3	Running Programs of Shared-Memory KLIC.....	60
4.6.3.1	Runtime Options for Shared-Memory KLIC.....	60
4.6.3.2	Known Bugs of Shared-Memory KLIC.....	60

<b>Data Type Index .....</b>	<b>61</b>
<b>Predicate, Method and Message Index.....</b>	<b>62</b>
<b>Module Index .....</b>	<b>65</b>
<b>Concept Index .....</b>	<b>66</b>

# Terms and Conditions for Use of ICOT Free Software

## 1. Purposes and Background of ICOT Free Software.

The Institute for New Generation Computer Technology (*ICOT*) had been promoting the Fifth Generation Computer Systems project under the commitment of the Ministry of International Trade and Industry of Japan (the *MITI*). Since April 1993, ICOT has been promoting the Follow-on project to the FGCS project. This follow-on project aims to disseminate and further develop FGCS technology. The FGCS project and the Follow-on project (collectively, the *Project*) have been aimed at creating basic technology for novel computers that realizes parallel inference processing as their core mechanism, and contributing toward the progress of computer science by sharing innovative knowledge and technology with the research community worldwide.

Innovative hardware and software parallel inference technology has been under development through the Project, which involves varieties of advanced software for experiments and evaluation. This software, being at a basic stage of research and development, should be disseminated widely to the research community.

According to the aims of the Project, ICOT has made this software, the copyright of which does not belong to the government but to ICOT itself, available to the public in order to contribute to the world, and, moreover, has removed all restrictions on its usage that may have impeded further research and development in order that large number of researchers can use it freely to begin a new era of computer science.

This program together with any attached documentation (collectively, the *Program*) is being distributed by ICOT free of charge as *ICOT Free Software*.

## 2. Free Use, Modification, Copying and Distribution

Persons wanting to use the Program (*Users*) may freely do so and may also freely modify and copy the Program. The term "modify," as used here, includes, but is not limited to, any act to improve or expand the Program for the purposes of enhancing and/or improving its function, performance and/or quality as well as to add one or more programs or documents developed by Users of the Program.

Each User may also freely distribute the Program, whether in its original form or modified, to any third party or parties, **PROVIDED** that the provisions of Section 3 (**NO WARRANTY**) will **ALWAYS** appear on, or be attached to, the Program, which is distributed substantially in the same form as set out herein and that such intended distribution, if actually made, will neither violate or otherwise contravene any of the laws and regulations of the countries having jurisdiction over the User or the intended distribution itself.

## 3. NO WARRANTY

The program was produced on an experimental basis in the course of the research and development conducted during the project and is provided to users as so produced on an experimental basis. Accordingly, the program is provided without any warranty whatsoever, whether express, implied, statutory or otherwise. The term "warranty" used herein includes, but is not limited to, any warranty of the quality, performance, merchantability and fitness for a particular purpose of the program and the nonexistence of any infringement or violation of any right of any third party.

Each user of the program will agree and understand, and be deemed to have agreed and understood, that there is no warranty whatsoever for the program and, accordingly, the entire risk arising from or otherwise connected with the program is assumed by the user.

Therefore, neither ICOT, the copyright holder, or any other organization that participated in or was otherwise related to the development of the program and their respective officials, directors, officers and other employees shall be held liable for any and all damages, including, without limitation, general, special, incidental and consequential damages, arising out of or otherwise in connection with the use or inability to use the program or any product, material or result produced or otherwise obtained by using the program, regardless of whether they have been advised of, or otherwise had knowledge of, the possibility of such damages at any time during the project or thereafter. Each user will be deemed to have agreed to the foregoing by his or her commencement of use of the program. The term "use" as used herein includes, but is not limited to, the use, modification, copying and distribution of the program and the production of secondary products from the program.

In the case where the program, whether in its original form or modified, was distributed or delivered to or received by a user from any person, organization or entity other than ICOT, unless it makes or grants independently of ICOT any specific warranty to the user in writing, such person, organization or entity, will also be exempted from and not be held liable to the user for any such damages as noted above as far as the program is concerned.



# 1 Introduction

This manual describes a portable implementation of KL1 called KLIC, developed at Institute for New Generation Computer Technology as a part of the Fifth Generation Computer national project of Japan and its follow-on project.

KL1 is a concurrent logic programming language based on Guarded Horn Clauses (GHC, in short). KL1 has very simple and concise syntax and semantics and yet provides very powerful features for concurrent computation.

KLIC compiles KL1 programs to C programs. A C compiler of the host system then compiles the C programs to relocatable objects, which will then be linked together with the runtime library of KLIC (see Section 4.1.3 [How KLIC Compiler Works], page 47). Thus, the system is independent from the hardware architecture of the host system. Also, the system is written so that only minimal features of Unix are used to assure portability.

## 1.1 Description of Predicates and Methods

### 1.1.1 Predicates and Methods

Unlike other logic programming language systems, KLIC provides two kinds of procedures, predicates and generic methods. Predicates define relations on their arguments and their semantics is fixed. Generic methods, (or methods, simply) are defined by *objects* they are applied to. Thus, their semantics depends on the object applied.

Sometimes the same operation is provided by both predicates and methods. For example, obtaining an element of a string can be done by either of the following two.

<code>string_element +String +Index -Element</code>	[Body Predicate on <code>builtin</code> ]
<code>element +String +Index -Element</code>	[Body Method on <code>string</code> ]

The former is a builtin predicate of the system. Predicate invocations are written as follows.

```
ModuleName:PredicateName(Arguments, ...)
```

In case of the predicate `string_element` mentioned above, it is defined as a builtin predicate and thus no module name is needed in its invocation. Thus, an invocation is written as follows.

```
string_element(String, Index, Element)
```

In general, a module name may come first with a colon before the predicate name. Some predicates do not have any arguments. In such cases, parentheses enclosing arguments are also omitted.

The latter is a generic method defined on the objects of class `string`. Method invocation is written as follows.

```
generic:MethodName(Object, OtherArguments, ...)
```

In case of the method `element` mentioned above, its invocation is written as follows.

```
generic:element(String, Index, Element)
```

The same operation may be effected using either a predicate or a method. For example, obtaining the third element (element number 2) of a string *S* into *E* can be done by either of the following invocations.

```
string_element(S, 2, E)
```

```
generic:element(S, 2, E)
```

Note that, while the predicate `string_element` is only for obtaining an element of a string, a generic method invocation can also be used to obtain an element of *similar* object. For example, an element of a vector (one-dimensional array) can also be obtained by the same invocation.

### 1.1.2 Messages

KL1 programs often consist of many *processes*. Processes often communicate one another using *streams*. Streams are actually lists of *messages*. Lists are made of cells called *cons* cells with two fields *car* and *cdr*, just as in Lisp or similar languages. Thus, when used as message streams, the car part of a cons cell contains the message and the cdr contains the rest of the stream.

Some standard features of the KLIC system are also provided as processes with message stream interface. An example of such message described in this manual follows.

```
putc +C [Message on C-like I/O]
```

This means that a message named `putc` is accepted as a message to a C-like I/O process interface stream. The message has one argument called *C* in this case.

To send a message to a message stream, the variable referring to the message stream should be instantiated with a cons cell whose car contains the message and cdr contains the rest of the stream. Thus, when *S* is a stream to C-like I/O and character with code 10 is to be output, the following unification should be made.

```
S = [putc(10) | T]
```

Here, the variable *T* is given the rest of the stream and thus any following messages to the stream should be sent to this variable.

### 1.1.3 Argument Modes

Arguments of predicates, methods or messages may have specific input/output mode. Input arguments are read by invocation of predicates or methods; the invocation will be suspended if any of the input arguments are left undefined. Output arguments are given a value by the invocation.

In the description of predicates and methods, input arguments are marked with a `+` and output arguments are marked with a `-`. Some arguments are not either read or given value. Such arguments are marked with a `?`.

## 1.2 Reporting Bugs and Sending Comments

Please report bugs and comments on the KLIC system and this document to the following mail address.

```
klic-bugs@icot.or.jp.
```

There is a mailing list for users of KLIC. This mailing list is used for announcement from the developers on known bugs, fixes or availability of new releases. The same mailing list can also be used for communication among users. To subscribe to the mailing list, please send your request to the following address.

```
klic-requests@icot.or.jp
```

## 2 KL1 Language

KL1 is a programming language for describing concurrent computation based on Guarded Horn Clauses (GHC, in short). GHC belongs to a family of languages called concurrent logic programming languages or committed-choice logic programming languages. Languages that belong to the family are, for example, Concurrent Prolog, Parlog, Fleng, Strand and Janus. These languages have simple and concise syntax and semantics and yet provide very powerful features for concurrent computation.

Here, a very rough and informal description of the KL1 language is given. More detailed and accurate specification are planned to be supplied in future (hopefully).

### 2.1 Basic Execution Mechanism

The following is an example of a small KL1 program that defines a part of quicksort program.

Example 1: Quicksort

```
:- module quicksort.

sort(X, Y) :- sort(X, Y, []).

sort([], Y, Z) :- Y = Z.
sort([P|X], Y, Z) :-
    partition(P, X, X1, X2),
    sort(X1, Y, [P|Y1]),
    sort(X2, Y1, Z).

partition(_, [], S, L) :-
    S = [],
    L = [].
partition(P, [W|X], S, L) :- W =< P |
    S = [W|S1],
    partition(P, X, S1, L).
partition(P, [W|X], S, L) :- W >= P |
    L = [W|L1],
    partition(P, X, S, L1).
```

The first line, `:- module quicksort` declares that this program module will be called `quicksort` (see Section 2.3 [Modules], page 6).

Execution of a KL1 program is a (possibly parallel) repetitive reduction of given *goals* using program *clauses*. Each clause has the following form.

*PredicateName*(*Argument pattern ...*) :- *Guard* | *Body*.

When a goal is to be reduced, clauses for the predicate of the goal will be inspected. For clauses with matching argument pattern, their guard parts are tested. All the clauses with matching argument pattern and satisfied guard conditions are candidates to be used in the reduction. Only one of them, *arbitrarily chosen*, will be used and the original goal will be replaced by the goals in the body of the clause chosen.

If no guard condition tests are required, the guard part along with the vertical bar can be omitted.

## 2.2 Predicates

*Predicates* of KL1 corresponds to subroutines of Fortran or functions of C. Predicates are defined by a collection of clauses with the same predicate name and the same number of arguments in their heads. Unlike in some other languages, predicates are identified not only by their names but also by their *arities* (numbers of arguments). To identify predicates with the same name but different arities, the notation *Predicate/Arity* is used in this manual.

In the example of the quicksort program, two predicates with the same name `sort`, with 2 and 3 arguments respectively, are defined (see Section 2.1 [Basic Execution Mechanism], page 5). Such predicates are referred to as `sort/2` and `sort/3` respectively.

The order of clauses defining a predicate does not affect the meaning. For example, a predicate for computing maximum of two integer values can be defined as follows.

```
max(X, Y, M) :- X >= Y | M = X.
max(X, Y, M) :- X <= Y | M = Y.
```

Exactly the same predicate can be defined by reversing the order of the clauses as follows.

```
max(X, Y, M) :- X <= Y | M = Y.
max(X, Y, M) :- X >= Y | M = X.
```

When a set of clauses are to be used *only when* another set of clauses are known not to be applicable, the keyword `otherwise` should be put in between the two sets of clauses. For example, the above ‘max’ predicate may be defined as follows.

```
max(X, Y, M) :- X >= Y | M = X.
otherwise.
max(X, Y, M) :- M = Y.
```

The meaning of the predicate is almost the same except that this version succeeds after unifying ‘M’ with ‘Y’ even when ‘X’ or ‘Y’ does not have an integer value, while the two previous versions will fail.

The `otherwise` directive specifies that clauses after the directive should not be applied unless all the clauses preceding the directive are known not to be applicable, even with any information (variable bindings) may become available afterwards. This feature should *not* be confounded with the `alternatively` directive, which specifies that clauses preceding the directive should be given priority to the clauses following (see Section 2.8 [Clause Preference], page 9).

## 2.3 Modules

KL1 provides *module* structure for dividing large programs into many modules. A module consists of one or more predicates. Definition of a module starts with a module declaration of the form `:- module Module`. Clauses defining predicates in the module will follow. The end of the file or another module declaration ends the definition of the module.

In the quicksort example, the first line:

```
:- module quicksort
```

declares that this program module is called `quicksort` (see Section 2.1 [Basic Execution Mechanism], page 5).

Predicates defined with the same name and arity but in different modules are considered to be different predicates. Thus, when necessary, the notation *Module:Predicate/Arity* is used to explicitly specify the module name.

## 2.4 Goals

A Goal is a unit of execution of KL1. Goals are associated with a predicate. A goal is reduced to zero or more simpler goals by applying one of the clauses defining the predicate.

Goals are written as:

*Predicate(Arguments, ...)*

or simply as the following.

*Predicate*

when the predicate has no arguments.

When the predicate is not in the same module, the syntax is either:

*Module:Predicate(Arguments, ...)*

or as the following.

*Module:Predicate*

For example, a module named `main` that uses the `quicksort` module might be defined as follows (see Section 2.1 [Basic Execution Mechanism], page 5).

Example 2: Module using `quicksort`

```
:- module main.

main :-
    X = [9,2,8,3,6,7,4,1,5],
    builtin:print(X),
    quicksort:sort(X, Y),
    builtin:print(Y).
```

Here, the body goal `quicksort:sort(X, Y)` is associated with the predicate `sort/2` of the module `quicksort`.

## 2.5 Initial Goal

All KLIC programs start from the initial goal `main:main`, i.e., the predicate `main` with no arguments defined in the module `main`. The example of the module `main` (see Section 2.4 [Goals], page 7) is an example of a main program.

Command line arguments are not passed to the initial goal. Predicates to access command line arguments are provided separately (see Section 3.5.7 [Predicate Interface], page 36).

## 2.6 Generic Objects

*Generic objects* provide a framework to extend the KL1 language with new data types and their operations. There are three kinds of generic objects, namely, *data objects*, *consumer objects* and *generator objects*.

Generic objects are created by pseudo-predicates `generic:new`. Generic data objects are similar to usual KL1 data. Operations on data objects are defined by their *generic methods*. Methods are invoked by pseudo-predicates `generic:Method`. Consumer and generator objects look like variables to normal KL1 programs and operations on them are implicit by unification.

Many of the standard types of KLIC, vectors and strings, for example, are actually implemented as generic data objects. For them, builtin predicates can also be used as aliases for generic methods. For example, `set_vector_element(Original, Index, NewElement, New)` means the same as `generic:set_element(Original, Index, NewElement, New)`.

### 2.6.1 Creating Generic Objects

Generic objects are created by the following pseudo-predicate.

```
generic:new(ClassName, Object, Args, ...)
```

*ClassName* should be a symbolic atom which names the object class. By this invocation, a new generic object is created and associated with *Object*. Parameters for creation can be specified by *Args*. The meaning of *Args* depends on each object class.

### 2.6.2 Guard Methods of Generic Data Objects

Clause selection depending on generic data objects can be done by calling guard methods. Guard methods have the following format.

```
generic:Method(Object, Input, ...):Output:...
```

*Input*, ... specify input arguments. If any of the input arguments are left undefined, this invocation will be suspended. *Output*:... specify output arguments, which are returned from the method. If some concrete value is specified as an *Output*, guard unification of the specified and returned values will be made. Some guard methods have no output arguments, in which case colon and following *Output* are omitted.

### 2.6.3 Body Methods of Generic Data Objects

Operations on generic data objects can be done by calling body methods. Body methods have the following format.

```
generic:Method(Object, Args, ...)
```

Unlike guard methods, input and output arguments are not syntactically distinguished. The method to be called can be determined in runtime. To do that, an alternative format is provided.

```
generic:generic(Object, Functor)
```

With this format, *Functor* should be (or become in runtime) a functor structure of the format *Method(Args, ...)*. Invocation will be suspended until *Functor* will become instantiated.

## 2.7 Priority Specification

Goals have execution priority associated with them. Execution priority is specified by a positive integer value. Goals with larger priority values are (usually) executed earlier than goals with smaller priority values. However, priority specifications are no more than suggestions and actual implementations may or may not strictly obey them.

Body goals can have execution priority specification in one of the following formats.

```
Goal@priority(AbsPrio)
Goal@lower_priority(RelPrio)
Goal@lower_priority
```

Here, *AbsPrio* and *RelPrio* should be a non-negative integer constant, or a variable which should be instantiated to a non-negative integer later. In the current implementation, negative priority values are interpreted as zero.

With the absolute priority specification, the goal with the specification will have the priority value specified by *AbsPrio*. With the relative priority specification, the goal will have priority less than the priority of the parent goal by the amount specified by *RelPrio*. The specification `Goal@lower_priority` has the same effect as `Goal@lower_priority(1)`. Goals without any priority specifications will have the same priority as their parents.

The highest possible priority is the largest possible integer value, which depends on host systems (see Section 3.2.2 [Integers], page 19). The initial goal `main:main` has the maximum priority possible for the host system (see Section 2.5 [Initial Goal], page 7).

## 2.8 Clause Preference

Predicates of KL1 may have nondeterminacy; more than one clause may be applicable. In such cases, preference among clauses may be specified using the **alternatively** directive.

When the keyword **alternatively** is put in between two sets of clauses, clauses preceding it are *preferred* to those following it. However, when clauses preceding the **alternatively** directive cannot be applied directly due to lack of information (insufficient instantiation of variable values), the clauses following it may be used. This feature is often useful in controlling speculative computation depending on progress of computation.

The feature is not to be confounded with the **otherwise** feature (see Section 2.2 [Predicates], page 6). For example, consider the following two predicates.

```
p(1, Y, R) :- R = a.
alternatively.
p(X, 2, R) :- R = b.

q(1, Y, R) :- R = a.
otherwise.
q(X, 2, R) :- R = b.
```

When the first argument is still undefined and the second is 2, the predicate `p` may return `b` to the third argument using its second clause. The predicate `q` will wait until the value of the first argument becomes available. Thus, if the first argument eventually becomes 1, the predicate `q` is guaranteed to return `a`, but the predicate `p` may return either `a` or `b`.

## 2.9 Shorthand Notation for Argument Pairs

KL1 programs often require passing two arguments as a pair to a predicate: one as input and the other as output. KLIC provides a shorthand notation for such cases.

### 2.9.1 Paired Arguments and their Expansion

The head and goals in both guard and body parts of a clause can have argument pairs specified by a single variable name attached to the head or goals by a hyphen character. We call such pseudo variable an *argument pair name*. An example is shown here.

$$p(X,Y)\text{-Pair} \text{ :- } q(X)\text{-Pair}, s(Z)\text{-Pair}, r(\text{Pair},Y), t(Z)\text{-Pair}.$$

The pseudo-variable **Pair** is an argument pair name. Such a clause is interpreted the same as the following clause.

$$p(X,Y,P0,P) \text{ :- } q(X,P0,P1), s(Z,P1,P2), r(P2,Y), t(Z,P2,P).$$

Occurrences of argument pair names attached to the head or goals by a hyphen character are interpreted as a pair of two different variables added at the end of the argument lists. In what follows, we call the two variables generated from an paired argument an *expanded pair*.

The second of an expanded pair of a goal is the same as the first of the expanded pair of the next goal with the same argument pair name. In the example above, **P1** appearing as the third argument of the goal of **q/3** also appears as the second argument of **s/3**, as originally they both have the same argument pair name **Pair**.

The first of an expanded pair in the head will be the same as the first of the expanded pair in the first goal in the clause with the same argument pair name. The second of an expanded pair in the head will be the same as the second of the expanded pair in the last goal with the same argument pair name.

In the above example, the first of the expanded pair **P0** in the head appears again as the second argument of the first goal calling **q/3**, and **P**, the second of the expanded pair in the head, appears again as the third argument of the last goal of **t/3**.

If the argument pair name appears only in the head, two variables of the expanded pair are unified in the body. For example, a clause:

$$p(X)\text{-Y} \text{ :- } q(X).$$

is expanded into the following.

$$p(X,Y0,Y) \text{ :- } Y0=Y, q(X).$$

An argument pair name may appear at a usual argument position rather than being attached to the head or goals, as does the first argument of the goal for **r/2** in the above example. In such a case, it is expanded to a single variable. This variable is the same as the second of the last expanded pair and is also the same as the first of the next expanded pair. Thus, in the above example, **Pair** appearing as the first argument of **r/2** is expanded into **P2**, which is the same as the third argument of **s/3** and the second argument of **t/3**.

Arbitrarily many argument pair names can be specified for a head or a goal. For example, a clause such as:

$$p\text{-X-Y} \text{ :- } q\text{-X}, r\text{-Y}, s\text{-Y-X}.$$

is interpreted as follows.

$$p(X0,X,Y0,Y) \text{ :- } q(X0,X1), r(Y0,Y1), s(Y1,Y,X1,X).$$

Sometimes, specifying normal arguments after some argument pair names is desirable. This can be done by connecting them with a plus (+) character. For example:

$$p\text{-X+Y} \text{ :- } q\text{-X+35}, r(Y), s\text{-Y-X}.$$



is interpreted as follows.

$p(X0,X,Y) :- q(X0,X1,35), r(Y), s(Y,X1,X).$

Note that the expansion rules for paired arguments described above are position sensitive for goals. However, this does *not* at all mean that the execution order of body goals are constrained anyhow.

Also note that the argument pair notation is no more than macro expansion of clauses. One predicate may have clauses some of which written in the argument pair notation and others in the usual notation.

### 2.9.2 Macros for Paired Arguments

To facilitate the usage of paired arguments, KLIC provides the following macros to be used in place of a goal.

$S <= M$       Expanded to  $S0 = [M|S1]$  where  $S0$  and  $S1$  are expanded pair for the argument pair name  $S$ .

$M => S$       Expanded to  $[M|S0] = S1$  where  $S0$  and  $S1$  are the expanded pair for the argument pair name  $S$ .

$S += E$

$S -= E$

$S *= E$

$S /= E$       Expanded to  $S1 := S0 + E0$  etc, where  $S0$  and  $S1$  are the expanded pair for the argument pair name  $S$ .

$S <== X$       Expanded to  $S1 = X$ , where  $S0$  and  $S1$  are the expanded pair for the argument pair name  $S$ .  $S0$  does not appear in the expansion; the original value of the paired argument for  $S$  will be lost, and the next occurrence of  $S$  will mean  $X$  instead. This feature is normally used with a non-paired occurrence of the argument pair name. For example:

..., p-S, q(S), S <== X, r-S, ...

means the following.

..., p(S0,S1), q(S1), S2 = X, r(S2,S3), ...

### 2.9.3 Usage of Paired Arguments

Some examples of typical usage of paired arguments are given here.

The following program is for summing up elements of a list of integers.

$sum(List,Sum) :- sum(List)+0+Sum.$

$sum([])-Acc.$

$sum([H|T])-Acc :- Acc += H, sum(T)-Acc.$

Here, the paired argument  $Acc$  plays the role of an accumulator.

The following program inverts the sign of the elements of a list of integers.

$inv(List,Inv) :- inv(List)+Inv-[].$

$inv([])-Inv.$

$inv([H|T])-Inv :- MH := -H, Inv <= MH, inv(T)-Inv.$

## 2.10 Inserting C Language Code Inline

The inline C code feature allows specifying C programs to be inserted in the object code within KL1 programs. This feature is somewhat similar to the `asm` statements of C.

Appropriateness of inserted C code totally depends on internal implementation schemes of the KLIC system, which may be altered in future. Thus, **general users are not recommended to use this feature.**

### 2.10.1 Inline Insertion at the Top of Files

At the top of a source file, strings to be inserted in the object C program can be specified in the following way.

```
:- inline:"C Program Text to be Inserted".
```

The specified text is inserted in the object C program after standard declarations and before any user-defined modules.

There can be any number of such inline specification. A typical example is as follows.

```
:- inline:"#include <stdio.h>"
```

As inserted C programs are written as string constants of KLIC, doublequote characters have to be escaped with a backslash character. A typical example is as follows.

```
:- inline:"#include \"myheader.h\""
```

It might also be a good idea to define macros and functions here, that are invoked from the inline code in clause guards.

### 2.10.2 Inline Insertion in the Guard

Inline insertion specification can also appear as a guard goal with one of the following forms.

```
inline:"C Program Text"
inline:"C Program Text":[ArgSpec, ...]
```

With either format, the C program text is literally inserted in the object code corresponding to the guard part, except that percent signs (%) in the program text string specify special formatting. The following table lists special format characters after percent signs and what they mean.

<code>digit</code>	The name of the C variable corresponding to the <code>digit</code> -th <i>ArgSpec</i> (zero origin). Note that only up to 10 such arguments are allowed.
<code>f</code>	The name of the C label to <code>goto</code> when this clause should fail or suspend.
<code>%</code>	The percent character itself, i.e., percent characters should be doubled. Be careful when you specify format strings for <code>printf</code> .

*ArgSpec* has one of the following formats.

#### *Variable+Type*

Specifies that the value of the variable is used within the inserted program text. Object code for synchronization with availability of the variable value and checking of the value type is generated by the compiler.

#### *Variable-Type*

Specifies that variable is given a value within the inserted program text. This has to be the first occurrence of the variable. The compiler assumes that, after executing the inserted code, the variable will have value of *Type*.

The Type field should be one of the following.

<b>any</b>	Anything, including uninstantiated variables
<b>bound</b>	Any bound value
<b>atomic</b>	An atomic value (a symbolic atom or an integer)
<b>int</b>	An integer
<b>atom</b>	An symbolic atom
<b>list</b>	A list structure
<b>functor</b>	A functor structure (including generic object)
<b>object</b>	A generic data object
<b>object(<i>Class</i>)</b>	A generic data object of <i>Class</i>

Values are referenced without any indirections for all types except for **any**. For an input (+) mode argument, the generated code makes sure that, before executing the inserted program text, the argument will have the value of the specified type directly, without indirect references. For an output (-) mode argument, the compiler assumes that, after executing the inserted program text, the variable will have the value of the specified type directly, without any indirect references, and uses that information for optimization. If you cannot be sure of this for output mode arguments, specify **any**, which may be less efficient but safe.

### 2.10.3 C-Level Representation of KL1 Terms

Note that C language types of the C variables and KL1 language types of corresponding KL1 values are not the same. All the C variables corresponding to a KL1 value have the type `q` which means almost nothing except that it occupies a single word. KL1 values are somehow encoded (with attached tag etc). For example, an integer 3 of KL1 is *not* represented by the bit pattern corresponding to integer 3 in of the language C.

This document is not intended to describe all the details of the data representation scheme of KLIC. Such description and programs depending on it will be obsoleted anyway by future revisions of the KLIC system. However, manipulation of integer values may be the easiest and useful in most C programs written inline. Thus, we'll describe macros for data conversion for integers. They are unlikely to be changed in future versions.

To obtain the integer value of a variable corresponding to an integer value of KL1, use the macro `intval(X)`. To obtain KL1 representation of integer in C, use the macro `makeint(N)`.

### 2.10.4 Examples

#### Example 1: Adding Two Integers

Two integers can be added by the following clause.

```
p(X,Y,Z) :- W := X+Y | Z = W.
```

The same function can be realized using the inline insertion feature as follows.

```
p(X,Y,Z) :-
```

```
inline:"%2 = makeint(intval(%0)+intval(%1));":
[X+int, Y+int, W-int] | Z=W.
```

The inserted text will be as follows.

```
x0 = makeint(intval(a0)+intval(a1));
```

Where variables `a0` and `a1` correspond to `X` and `Y`, and `x0` to `W` in the KL1 program. Note that the `Z` and `W` are unified in the body.

### Example 2: Comparing Two Integers

Two integers can be compared by the following clause.

```
p(X,Y) :- X > Y | ...
```

The same function can be realized using the inline insertion feature as follows.

```
p(X,Y) :-
inline:"if (intval(%0) <= intval(%1)) goto %f;":
[X+int, Y+int] | ...
```

The inserted text will be as follows.

```
if (intval(a0) <= intval(a1)) goto p_2_interrupt;
```

Where variables `a0` and `a1` correspond to `X` and `Y` in the KL1 program, and `p_2_interrupt` is a label automatically generated by the compiler.

### 2.10.5 Some Hints on Using the Inline C Code Feature

- Whenever possible, avoid using the inline feature. Revisions of the KLIC system may obsolete your code.
- If multiple lines are to be inserted consecutively, specify all of them in one single inline specification. Otherwise, they might be interleaved by other code for the guard. Newlines are allowed within the inserted program text.
- Do not forget to prefix doublequotes and backslashes with a backslash. Do not forget also to write two percent signs to insert one. If you would like to say hello to the world, you should write such a program as follows.

```
hello :-
inline:"printf(\"Hello, world\\n\\n\");" |
...
```

Note the backslashes before doublequotes within the inlined code and doubled backslash before `n`. If you put only one backslash before `n`, it will become a newline code after inline expansion; it will appear within a string constant in the expanded C program. It fortunately works the same in this case, except that some C compilers may generate a warning message.

- If your program with inline code does not work as you expect it to, the best way to find the problem may be to look into the C code generated.

## 3 Builtin and Library Features

This chapter describes builtin and library features of KLIC.

### 3.1 Common Operations

Some predicates are used commonly for all the data types or are independent from any data types.

#### 3.1.1 Unification

`= ?X ?Y` [Guard Predicate on `builtin`]  
Checks whether  $X$  and  $Y$  are unifiable without giving values to variables outside the clause.

`= ?X ?Y` [Body Predicate on `builtin`]  
Unifies  $X$  and  $Y$ . If  $X$  has no value yet and  $Y$  already has some defined value, the value of  $Y$  is given to  $X$ . If  $Y$  has no value and  $X$  has some, the reverse takes place. If both do not have values yet, two variables are made to mean the same variable. If both have values, they are matched. If both are the data structures of the same kind, this unification operation is made recursively to corresponding elements of two structures.

#### 3.1.2 Synchronization

`wait +X` [Guard Predicate on `builtin`]  
Waits until instantiation of  $X$ .

#### 3.1.3 Comparison and Hashing

`compare +X +Y -R` [Guard Predicate on `builtin`]  
Compares  $X$  and  $Y$ , and returns the result in  $R$ . The result is an integer value less than, equal to, or greater than 0, when  $X$  is less than, equal to, or greater than  $Y$ , respectively.

The comparison is made by the standard order. This predicate can compare data of any types. If both  $X$  and  $Y$  are of the same numeric type, normal numerical comparison is made. Note that integer and floating point numbers are *not* of the same type; their comparison may not be meaningful. Two strings are compared in (so-called) dictionary order.

The order of any two data of different types is somehow defined by the system. However, the ordering is guaranteed to be kept only within a single executable program. If some data sequence is saved into a permanent file using the ordering provided by this predicate, the same program recompiled or linked with some other programs may or may not recognize the sequence as ordered. Different programs, of course, may use different ordering.

Both  $X$  and  $Y$  have to be instantiated enough for making the comparison. For example '`f(V) @< f(W)`' will suspend if not both ' $V$ ' and ' $W$ ' are instantiated. On

the other hand, ‘ $f(1,V) @< f(2,W)$ ’ will succeed immediately, as the order can be determined without looking into values of ‘ $V$ ’ or ‘ $W$ ’.

The absolute value of the result  $R$  may have some meaning for certain data types. When comparing two strings, the absolute value of the result is one more than the index of the first differing element (a la `strcmp` of C).

Some generic objects may not implement their comparison methods, and, in such cases, their comparison will result in a fatal error.

**@< +X +Y** [Guard Predicate on builtin]  
**@=< +X +Y** [Guard Predicate on builtin]  
**@>= +X +Y** [Guard Predicate on builtin]  
**@> +X +Y** [Guard Predicate on builtin]  
 Compares  $X$  and  $Y$  with the standard order. If the condition is not satisfied, the invocation of the predicate fails.

**\= +X +Y** [Guard Predicate on builtin]  
 Compares  $X$  and  $Y$  and succeeds if and only if their principal functors are different. For atomic values, it means that they are different; for functor structures, it means that either they have different functor names or different arity. For generic objects, the predicate succeeds when two objects are of different classes.  
 Note that ‘ $f(a) \backslash= f(b)$ ’ fails, as the two terms have the same principal functor. Note also that, floating point numbers are generic objects and thus ‘ $X \backslash= Y$ ’ fails for any two floating point numbers, as they are objects of the same class.

**hash +X -H** [Guard Predicate on builtin]  
 Computes the hash value of  $X$  and returns it in  $H$ . The hash value is a non-negative integer value.  
 Hashing function may look into elements of structured data recursively.  $X$  has to be instantiated enough to compute hash value. Some generic objects may not implement hash methods, and, in such cases, their hash value becomes a constant.

### 3.1.4 Execution Status

**current\_priority -P** [Guard Predicate on builtin]  
 Returns the priority value of the reduced goal to  $P$ . See Section 2.7 [Priority Specification], page 8, for further details on the priority mechanism.

**current\_node -Node -NumNodes** [Body Predicate on builtin]  
 On a parallel implementation, the predicate returns the processor number executing the predicate in  $Node$  and the total number of (pseudo-) processors available in  $NumNodes$ . Processor numbers have zero origin. Thus, the maximum value returned to  $Node$  is one less than the value returned to  $NumNodes$ . On a sequential implementation, 0 is returned to  $Node$  and 1 to  $NumNodes$ .

### 3.1.5 Debugging

**unbound ?X -Result** [Body Predicate on builtin]  
 Checks whether  $X$  is already bound to some concrete value or not and returns the *Result*.

If the toplevel of  $X$  is already defined, *Result* is unified with a single-element vector of the form  $\{X\}$ . When  $X$  is bound to a structured value, its elements may or may not be bound yet.

If  $X$  is not bound yet, *Result* is unified with a three element vector of the form  $\{Addr1, Addr2, X\}$ , where *Addr1* and *Addr2* are integers indicating the current address of the variable  $X$  somehow. Note that variable addresses may change in time by garbage collection, automatic data migration in parallel implementations or any such low level implementational reasons; they are no more than debugging hints.

**Do not use this predicate in normal application programs.** Unlike the `var/1` feature of sequential Prolog, variables once judged as unbound can be bound in the next instance on parallel implementations. Thus, usage of this predicate should be restricted to programs that have to go into low level details of the system implementation, such as debugging tools.

## 3.2 Atomic Data

KLIC provides two kinds of atomic data types, numerical and symbolic.

For numerical data, KLIC provides integer and floating point number data types and operations to manipulate them. Floating point numbers are implemented as generic objects and thus actually are not an atom.

Note that implicit type conversions between integer and floating point data are never made. Integer numbers and floating point numbers are treated completely separately.

Whether given data is atomic or not can be tested by the following guard predicates.

**atomic +X** [Guard Predicate on `builtin`]  
 Tests whether  $X$  is atomic or not. Floating point numbers are *not* judged as atomic by this predicate.

### 3.2.1 Symbolic Atoms

*Symbolic atoms* are atomic data objects that give names to notions. Symbolic atoms with the same name are the same and with different names are different.

#### 3.2.1.1 Notation of Symbolic Atoms

The notation of symbolic atoms is similar to Edinburgh Prolog, which is one of the following.

- A lower case letter followed by a sequence of any number (including zero) of letters, digits or underlines.

Examples:

```
icot    kl1    a_symbolic_atom_with_a_long_name
```

- A sequence of special characters (some of `~`, `+`, `-`, `*`, `/`, `\`, `^`, `<`, `>`, `=`, `'` (backquote), `:`, `.`, `?`, `@`, `#`, `$`, `&`).

Examples:

```
+    >=    :-    =:=
```

- A sequence of any characters quoted by single quotes. If single quote characters are to be included, they should be doubled or escaped by a backslash.

Examples:

```
'Hello world'      'an atom with \'singlequotes\' in it'
```

- Special one-character atoms. There are three of them, which are `!`, `|` and `;`. Also, `|` has a special meaning in list notation (See Section 3.3.2.1 [Notation of Lists], page 25).
- A special atom `[]`, which usually is used to represent ends of lists (see Section 3.3.2 [Lists], page 24). Spaces can be in between `[` and `]`.

Important differences with Edinburgh Prolog syntax are the following.

- A vertical bar (`|`) means a one-character atom. Even when it is used as an operator, it is *not* treated the same as a semicolon (`;`) but as a different atom.
- A pair of curly braces (`{}`) does not stand for a symbolic atom. It means a vector with no elements (see Section 3.3.3.1 [Notation of Vectors], page 27).

### 3.2.1.2 Operations on Symbolic Atoms

Whether a given data object is a symbolic atom or not can be tested by the following guard predicate.

**atom +X** [Guard Predicate on **builtin**]  
Tests whether *X* is a symbolic atom or not.

To maintain the uniqueness of atoms, the system gives a unique number to each atom and maintains the association between atom name strings and atom numbers. Association of symbolic atoms and their names can be known by the following predicates defined in the module `atom_table`.

**make\_atom +String -Atom** [Predicate on **atom\_table**]  
When given a *String*, returns *Atom* with that name. If such an atom does not exist, a new atom is registered.

**atom\_number +Atom -Number** [Predicate on **atom\_table**]  
Internal serial number for *Atom* is returned to *Number* as an integer value.

**get\_atom\_string +Atom -String** [Predicate on **atom\_table**]  
The name string of *Atom* is returned to *String*.

**intern +String -Result** [Predicate on **atom\_table**]  
The same as `atom_table:make_atom`, except that the returned value is a functor structure of the form `normal(Atom)`.

**get\_atom\_name +Atom -Result** [Predicate on **atom\_table**]  
The same as `atom_table:get_atom_string`, except that the returned value is a functor structure of the form `normal(String)`.

Although symbolic atoms are associated with their name strings, do *not* use them for string manipulation. String data objects provide much more functionality and better performance (see Section 3.3.4 [Strings], page 28).



### 3.2.2 Integer Atoms

KLIC provides integer data with usually 28 or 60 bits as its basic standard feature. The width depends on the C compiler you use. It is 4 bits shorter than the width of type `long int`.

`integer +X` [Guard Predicate on `builtin`]  
 Tests whether *X* is an integer atom.

#### 3.2.2.1 Notation of Integers

KLIC provides several ways to denote integer constants.

- Usual decimal notation: optional minus sign followed by a sequence of decimal digits. Examples: `'123'`, `'-35'`.
- Based notation: optional minus sign followed by a sequence of decimal digits specifying the base (1 to 36), an apostrophe, and then a sequence of digits of the base, that are digits and alphabets (case insensitive). Examples: `'2'1010'`, `'16'0D0a'`. Value of an integer with base 1 is the number of ones in the digit sequence; for example `'1'10110'` means 3.
- Character code notation: optional minus sign followed by a digit 0, an apostrophe and a character. Examples: `'0'a'` means the character code of lowercase letter `a`.

The above listed constant notations can be used in both KL1 programs and KL1 data read in by Prolog-like I/O interface (see Section 3.6.2 [Input and Output with Prolog-like Interface], page 39).

The following are also allowed in KL1 programs for compatibility with PIMOS system on PIM machines.

- Based notation: optional minus sign followed by a sequence of decimal digits specifying the base (1 to 36), a sharp sign, and then a character string of digits of the base, that are digits and alphabets (case insensitive), surrounded by doublequotes. Examples: `'2#"1010"'`, `'16#"0D0a"'`.
- Character code notation: optional minus sign followed by a sharp sign and a character enclosed within doublequotes. Examples: `'#"a"'` means the character code of lowercase letter `a`.

#### 3.2.2.2 Integer Arithmetics

`:= -Var +Expr` [Guard Predicate on `builtin`]  
`:= -Var +Expr` [Body Predicate on `builtin`]  
 Computes the value of the integer expression *Expr*, and unifies it with *Var*. The following operators are available in the expression.

<code>X + Y</code>	Addition.
<code>+ X</code>	No operation. <i>X</i> is the result.
<code>X - Y</code>	Subtraction.
<code>- X</code>	Sign inversion.
<code>X * Y</code>	Multiplication.

$X / Y$	Integer division.
$X \bmod Y$	Modulo.
$\backslash(X)$	Bit-wise complement.
$X /\backslash Y$	Bit-wise logical AND.
$X \backslash/ Y$	Bit-wise logical OR.
$X \text{ xor } Y$	Bit-wise exclusive OR.
$X \ll Y$	Left shift.
$X \gg Y$	Logical right shift.
<code>int(X)</code>	Conversion from floating point to integer. $X$ is a floating point expression (see Section 3.2.3.3 [Floating Point Arithmetics], page 21) and its result is rounded to an integer value.

Arithmetical overflows are ignored, that is, all the arithmetics are done modulo  $2^{28}$  or  $2^{60}$  depending on the C compiler used. C compilers with 32-bit `long int` give 28-bit KLIC integers and those with 64-bit `long int` give 60-bit KLIC integers.

This predicate is available in both guards and bodies of clauses.

If any of the operands in the expression are uninstantiated, the computation will be suspended until they all get instantiated.

Any operands in the expression can be an expression recursively. However, operands written as a variable in the program should *not* be instantiated to a compound term such as ‘3 + 5’. They should be instantiated only to an integer. Otherwise, a type mismatch error will be generated.

### 3.2.2.3 Integer Comparison

Comparison of integer data can be made using the predicates described here. More general comparison predicate is also provided (see Section 3.1.3 [Comparison and Hashing], page 15), but predicates and methods described here are more efficient when the operands are known to be integers.

<code>&gt; +X +Y</code>	[Guard Predicate on builtin]
<code>&gt;= +X +Y</code>	[Guard Predicate on builtin]
<code>:= +X +Y</code>	[Guard Predicate on builtin]
<code>=\= +X +Y</code>	[Guard Predicate on builtin]
<code>=&lt; +X +Y</code>	[Guard Predicate on builtin]
<code>&lt; +X +Y</code>	[Guard Predicate on builtin]

Perform arithmetical comparison of two integer arguments. Use `:=` and `=\=` for equality and non-equality checks. Each side of the comparison can be an arithmetical expression. The same set of operators as in `:=` can be used.

### 3.2.3 Floating Point Numbers

Floating point numbers with precision of 64 bits are provided as generic objects. The following method and predicate tell whether given data is a floating point number or not.

`float +X` [Guard Method on `float`]  
`float +X` [Guard Predicate on `builtin`]  
 Tests whether *X* is a floating point number.

### 3.2.3.1 Notation of Floating Point Numbers

Floating point numbers have the following constant notation syntax.

*sign integral . fraction e sign exponent*

where *integral*, *fraction* and *exponent* are sequence of decimal digits. *sign* is either +, - or empty (meaning +). The exponent part, that is, character *e*, *sign* and *exponent*, may be omitted altogether.

The following are examples of floating point number constants.

3.14159   -6.02e23   1234.5678e-25

### 3.2.3.2 Creating New Floating Point Numbers

New floating point numbers can be created by the following. Predicates for floating point arithmetics described in Section 3.2.3.3 [Floating Arith], page 21, also create floating point numbers as the result of arithmetical operations.

`new -Float +Init` [Object Creation on `float`]  
 A new floating point number is created and unified with *Float*. The argument *Init* should be an integer specifying the value of the floating point number. For example, `'generic:new(float, F, 3)'` unifies *F* with 3.0.

### 3.2.3.3 Floating Point Arithmetics

`$ := -Var +Expr` [Body Predicate on `builtin`]  
 Computes the value of the floating point expression *Expr*, and unifies it with *Var*. The following operators are available in the expression.

*X* + *Y*      Addition.

*X* - *Y*      Subtraction.

*X* \* *Y*      Multiplication.

*X* / *Y*      Division.

`pow(X, Y)`  
               *Y* to the power of *X*.

`sin(X)`, `cos(X)`, `tan(X)`  
               Trigonometric functions on *X*.

`asin(X)`, `acos(X)`, `atan(X)`  
               Inverse trigonometric functions on *X*.

`sinh(X)`, `cosh(X)`, `tanh(X)`  
               Hyperbolic functions on *X*.

`exp(X)`      Exponential function.

`log(X)`      Natural logarithm.

<code>sqrt(X)</code>	Square root.
<code>ceil(X)</code>	Ceiling function (rounding toward positive infinity).
<code>floor(X)</code>	Flooring function (rounding toward negative infinity).
<code>float(X)</code>	Conversion from an integer to a floating point number. <i>X</i> is an integer expression (see Section 3.2.2.2 [Integer Arithmetics], page 19) and its result is converted to a floating point number.

This predicate is available in both guards and bodies of clauses.

If any of the operands in the expression are uninstantiated, the computation will suspend until they all get instantiated.

Any operands in the expression can be an expression recursively. However, operands written as a variable in the program should *not* be instantiated to a compound term such as `'3.0 + 5.0'`. They should be instantiated only to a floating point number. Otherwise, a type mismatch error will be generated.

Operations listed above are also provided as generic methods on floating point number.

<code>add +X +Y -R</code>	[Body Method on float]
<code>subtract +X +Y -R</code>	[Body Method on float]
<code>multiply +X +Y -R</code>	[Body Method on float]
<code>divide +X +Y -R</code>	[Body Method on float]
<code>pow +X +Y -R</code>	[Body Method on float]
<code>sin +X -R</code>	[Body Method on float]
<code>cos +X -R</code>	[Body Method on float]
<code>tan +X -R</code>	[Body Method on float]
<code>asin +X -R</code>	[Body Method on float]
<code>acos +X -R</code>	[Body Method on float]
<code>atan +X -R</code>	[Body Method on float]
<code>sinh +X -R</code>	[Body Method on float]
<code>cosn +X -R</code>	[Body Method on float]
<code>tanh +X -R</code>	[Body Method on float]
<code>exp +X -R</code>	[Body Method on float]
<code>log +X -R</code>	[Body Method on float]
<code>sqrt +X -R</code>	[Body Method on float]
<code>ceil +X -R</code>	[Body Method on float]
<code>floor +X -R</code>	[Body Method on float]

These methods perform arithmetic operations, specified by the method name, on given operand(s), and return the result in *R*.

### 3.2.3.4 Floating Point Comparison

Comparison of floating point data can be made by the predicates described here. More general comparison predicate is also provided (see Section 3.1.3 [Comparison and Hashing], page 15), but predicates and methods described here are more efficient when the operands are known to be floating point numbers.

<code>\$&gt; +X +Y</code>	[Guard Predicate on builtin]
<code>\$&gt;= +X +Y</code>	[Guard Predicate on builtin]

<code>\$:= +X +Y</code>	[Guard Predicate on builtin]
<code>\$=\ +X +Y</code>	[Guard Predicate on builtin]
<code>\$=&lt; +X +Y</code>	[Guard Predicate on builtin]
<code>\$&lt; +X +Y</code>	[Guard Predicate on builtin]

These predicates perform arithmetical comparison of two floating point arguments. Use `:=` and `=\` for equality and non-equality checks (although they may not be much meaningful for floating point numbers). Each side of the comparison can be a floating point arithmetical expression. The same set of operators as in `$:=` can be used.

**Bug Caution** The current version (1.510) has problems with expressions with operators in these predicates. Only simple variables and constants can be used.

Comparison of floating point numbers can also be made by the methods described below.

<code>less_than +X +Y</code>	[Guard Method on float]
<code>not_greater_than +X +Y</code>	[Guard Method on float]
<code>not_less_than +X +Y</code>	[Guard Method on float]
<code>greater_than +X +Y</code>	[Guard Method on float]
<code>equal +X +Y</code>	[Guard Method on float]
<code>not_equal +X +Y</code>	[Guard Method on float]

These methods test whether *X* is less than *Y* or not, etc.

### 3.3 Structured Data

Structured data objects consist of zero or more elements.

#### 3.3.1 Functor Structures

Functor structures are structures with given name and one or more elements, which can be of any type. Functors are conveniently used for representing data structures whose sizes are known beforehand. Functors correspond to record structures of C-like languages.

##### 3.3.1.1 Notation of Functors

Functor constants can be written by the name of the principal functor, a left parenthesis, elements separated by commas, and finally a right parenthesis. Functor names have the same syntax as symbolic atoms. The principal functor name and the following left parenthesis should *not* be separated by space characters or any other punctuation symbols. Elements can be of any type, including variables or functors themselves.

Examples:

```
f(a, 3)    'a recursive functor structure'(X, 'child functor'(Y))
```

##### 3.3.1.2 Operations on Functors

Predicates for manipulation of functor structures are provided as builtin predicates and in the module `functor_table`, as listed in this section.

In the current implementation, all the body builtin predicates listed here are actually implemented as macros expanded to predicates of the module `functor_table`. This implementation scheme may be changed in future releases.

**functor** *+X -Functor -Arity* [Guard Predicate on **builtin**]  
**functor** *+X -Functor -Arity* [Body Predicate on **builtin**]

*X* is a functor with the principal functor whose name being *Functor* and arity *Arity*. These predicates can be used for obtaining the the name and/or the arity of principal functors. The guard predicate version can also be used for testing that *X* has the name *Functor* and/or the arity *Arity*. Any instantiated data that are not functor structures, i.e., atomic data, strings, vectors and so on, have zero as their arities and themselves as their principal functor names. Note that list structures consist of functors *./2*.

This predicate cannot be used to create a new functor.

**arg** *+Pos +Term -Arg* [Guard Predicate on **builtin**]  
**arg** *+Pos +Term -Arg* [Body Predicate on **builtin**]

The *Pos*-th argument of *Term* is *Arg*. Arguments are numbered from 1. The guard version simply fails if *Pos* is out of range. As all the data structures except for functor structures have no arguments, this predicate always fails for them.

**new\_functor** *-Functor +Atom +Arity* [Body Predicate on **builtin**]

A functor structure with its principal functor with name *Atom* and arity *Arity* is returned to *Functor*. Arguments of the created functor are initiated with integer 0.

**setarg** *+Pos +Fnct ?NewE -NewFnct* [Body Predicate on **builtin**]

**setarg** *+Pos +Fnct ?OldE ?NewE -NewFnct* [Body Predicate on **builtin**]

A new functor structure that is different from *Fnct* with only one argument at *Pos* is created and returned to *NewFnct*. The element with index *Pos* of *NewFnct* will be *NewE*. For five argument versions, the original argument at *Pos* will be returned to *OldE*.

**=..** *-NewFnct +List* [Predicate on **functor\_table**]

A new functor structure is created and returned to *NewFnct*. The name of the principal functor is specified by the first element of *List*, which should be a symbolic atom, and the arguments are specified by the rest of *List*. If *List* has only one element, that element is returned to *NewFnct*.

This predicate can *not* be used for decomposing a functor structure to a list.

### 3.3.2 Lists

Lists are arbitrarily long sequences of any data objects. In KL1, List structures are made up of functor structures *./2*, that is, functor structures with their name *.* and arity two. List structures are composed of possibly many of these functor structures (sometimes called *cons cells*).

The first element of the cons cell, sometimes called the *car* of the cell, represents the first element of the list. The second element, the *cdr* of the cell, represents the rest of the list. Termination of the list is indicated by a symbolic atom *[]* being the *cdr*.

Whether a given argument is a list or not can be tested by the following guard predicate.

**list** *+X* [Guard Predicate on **builtin**]

Tests whether *X* is a cons cell. Note that, despite its name, this predicate fails for a null list *[]* for a historical reason.

Incrementally instantiated list structures are conveniently used as message streams.

### 3.3.2.1 Notation of Lists

As in Lisp, lists of KL1 are constructed using *cons* data structures, which is actually a functor structure `./2`.

The basic notation for lists is `[Car | Cdr]`, which consists of the first element *Car* and the tail of the list *Cdr*. This means exactly the same as `.(Car, Cdr)`. An empty list is represented by an atom `[]`.

If *Cdr* happens to be empty, that is, when the list consists only of one element *Car*, such a list can be written as `.(Car, [])` or `[Car | []]`, or, alternatively, as `[Car]`. That is, the sequence `| []` at the tail of a list can be omitted.

Lists with its car being *Car* and its cdr being a list `[Cadr, ...]` is `[Car | [Cadr, ...]]`, which can be abbreviated as `[Car, Cadr, ...]`. For example, a list consisting of four elements, *first*, *second*, *third* and *fourth* can be written as `[first, second, third, fourth]`.

A list consisting of four or more elements, but with the first four elements being *first*, *second*, *third* and *fourth*, can be written as `[first, second, third, fourth | Rest]`. Here, the variable *Rest* corresponds to the list beginning with the fifth element, or an empty list if the whole list had only four elements.

Note that, unlike in Edinburgh Prolog, the character sequence `,...` can *not* be used in place of `|`.

### 3.3.2.2 Manipulation of Message Streams

A stream merger is a process that takes multiple message streams represented as lists of messages as input, and passes all the messages from all the input streams to a single output stream also represented as a list.

The output consists of all the messages in the inputs with duplicates preserved. When two messages are ordered in one of the input streams, their order is also preserved in the output. When messages are from different input streams, their order in the output is unpredictable. The order may differ in one execution of the same program from another. The behavior of mergers is thus nondeterministic.

For example, when there are two input streams `[1, 2, 3]` and `[a, b, c]`, the output can be something like `[1, 2, a, b, 3, c]` or `[1, a, 2, b, c, 3]`, but will never be `[1, a, 3, b, c, 2]`.

A binary (two-input) stream merger can be defined in KL1 as follows.

```
merge([M|In1], In2, Out) :- Out=[M|OutT], merge(In1, In2, OutT).
merge(In1, [M|In2], Out) :- Out=[M|OutT], merge(In1, In2, OutT).
merge([], In2, Out) :- Out=In2.
merge(In1, [], Out) :- Out=In1.
```

- The first clause forwards one message coming from the first input stream to the output stream. The first input stream is the first argument and the output stream is the third argument of the predicate. It then calls the predicate `merge/3` recursively for repetitive execution.
- The second clause does the same for the second input stream.

- The third clause is used when the first input stream has no more messages in it. In this case, the second input stream is directly connected to the output. As there are no more messages to merge from the first input stream, the result of the merging should be the same as the second input stream.
- The fourth clause provides the corresponding feature when the second input stream has no more messages in it.

When messages come from both the first and the second at the same time, either the first or the second clause is arbitrarily chosen. This is the source of the nondeterminacy of the merger.

Although binary mergers are easy to define in KL1, defining a merger with arbitrarily many input streams is not so easy. It is also desirable to add new input streams dynamically, which makes it still harder. Also, mergers are used quite frequently in KL1 programs and thus should be quite efficient. Thus, the KLIC system provides a merger as one of its standard feature.

A new merger can be created by the following pseudo-predicate.

**new ?Input ?Output** [Object Creation on merge]  
 A new merger with single input stream is created. Its input stream is *Input* and its output is *Output*.

The merger process created by the above pseudo-predicate does not actually start any merging immediately after its creation. It only forwards the messages from *Input* to *Output*, without changing the order.

To add a new input stream to a merger, unify the input with a vector whose elements are input streams. For example, if you need a binary merger, do the following.

```
generic:new(merge, Input, Output),
Input = {In1, In2}
```

This means the same as the following.

```
generic:new(merge, {In1, In2}, Output)
```

After doing the above, the merger will merge messages from two input streams, *In1* and *In2*, to the output stream *Output*.

Input streams to a merger can be added not only immediately after its creation but at any time on demand. Two more input streams are added, for example, by the following.

```
In2 = {In2A, In2B, In2C}
```

After this, the merger will have four input streams, *In1*, *In2A*, *In2B* and *In2C*.

When one of the input streams is no longer needed, that input stream can be simply closed, by unifying it with an atom [].

The size of the vector unified with an input stream can be arbitrarily large or small. When it has only one element, the number of input streams will not be changed. When the vector has no elements, unifying with it has the same effect as closing the stream.

The output stream will be closed, i.e., the tail of the output list is unified with [], when all the input streams have been closed.



Here are some clues in using the merger.

- Messages merged can be a data structure containing unbound variables. Such messages are sometimes called *incomplete messages*. Incomplete messages are convenient for constructing a server-client process structure. Giving values to variables in messages can be used for communicating backwards from the server to the client.
- Merging may look deterministic on sequential implementations. Do never rely on it. It will become really nondeterministic on parallel implementations.

### 3.3.3 Vectors

Vectors are fixed-length one-dimensional array of KL1 data. The length of a vector is determined on its creation. Elements can be any KL1 data and can even be left undefined when the data structure is created.

Elements are indexed by an integer beginning from 0. For example, a vector with 3 elements has elements numbered 0, 1 and 2.

#### 3.3.3.1 Notation of Vectors

Vectors can be denoted by a comma-separated list of elements in a pair of curly braces.

`{ 1, a, f(b), X }`

A null vector (vectors with no elements at all) is denoted only by a pair of curly braces.

`{}`

Note that curly braces are used in a way completely different from Edinburgh Prolog, where `{}` means an atom and `{...}` means a functor structure `{>((...))}`.

#### 3.3.3.2 Creating New Vectors

In addition to the notation described above, vectors can be dynamically created during program execution. The following predicate can be used to create a new vector.

<code>new -Vector +Init</code>	[Object Creation on <code>vector</code> ]
<code>new_vector -Vector +Init</code>	[Body Predicate on <code>builtin</code> ]

A new vector is created and returned to *Vector*.

If the argument *Init* is an integer, it specifies the number of elements. The elements are initialized with integer 0 in this case. For example, `'generic:new(vector, V, 2)'` creates a vector `'{0, 0}'` and returned it to *V*.

If *Init* is a list, the newly created vector is initiated by the elements of the list. Naturally, the number of elements of the vector becomes the same as the length of the list. For example, `'generic:new(vector, V, [a, b, c])'` creates a vector `{a, b, c}` and unifies it with *V*.

#### 3.3.3.3 Predicates on Vectors

<code>vector +Vector -Length</code>	[Guard Method on <code>vector</code> ]
<code>size +Vector -Length</code>	[Body Method on <code>vector</code> ]
<code>vector +Vector -Length</code>	[Guard Predicate on <code>builtin</code> ]

Tests whether *Vector* is a vector object (if called in guard) and returns the number of elements in *Length*.

`element +Vector +Index -Element` [Guard Method on `vector`]  
`element +Vector +Index -Element` [Body Method on `vector`]  
`vector_element +Vector +Index -Element` [Guard Predicate on `builtin`]  
`vector_element +Vector +Index -Element` [Body Predicate on `builtin`]

An element with index *Index* of the vector *Vector* is unified with *Element*. The index is zero origin.

`set_element +Original +Index ?NewElement -New` [Body Method on `vector`]  
`set_vector_element +Original +Index` [Body Predicate on `builtin`]  
`?NewElement -New`

A new vector is unified with *New*. The new vector has the same elements as the *Original*, except that the *Index*'th element is updated to *NewElement*. The original vector is left untouched. The index is zero origin.

`set_element +Original +Index ?Element` [Body Method on `vector`]  
`?NewElement -New`  
`set_vector_element +Original +Index ?Element` [Body Predicate on `builtin`]  
`?NewElement -New`

A new vector is unified with *New*. The new vector has the same elements as the *Original*, except that the *Index*'th element is updated to *NewElement*. The original vector is left untouched. The index is zero origin. The original *Index*'th element is returned to *Element*.

`split +Original +At -Lower -Upper` [Body Method on `vector`]

The vector *Original* is split at the index position *At* and the resultant two vectors are unified with *Lower* and *Upper*. *At* has to be a non-negative integer less than or equal to the size of the original vector. *Lower* will have elements with indices between 0 and *At*-1, inclusive. Elements with indices between *At* and up will be included in *Upper*.

`join +Lower +Upper -Joined` [Body Method on `vector`]

Two vectors *Lower* and *Upper* are concatenated together to make a new vector *Joined*.

In KLIC, creating a new vector differing with an existing one by only a single element is implemented with constant time and space overhead, regardless of the size of the vector, by using multiversion array representation.

### 3.3.4 Strings

Strings are one dimensional arrays of integers in restricted range. The current version provides only strings of 8-bit elements which has elements of the range 0 through 255. They are convenient for representing character strings. Strings with elements of different sizes are planned in future.

Unlike in Edinburgh Prolog, strings are *not* notational convention for lists of character codes. They are of its own data type.

#### 3.3.4.1 Notation of Strings

In KLIC, character string constants should be denoted by sequence of characters surrounded by a pair of doublequotes, as follows.

"A string of the characters written here"

The following escape sequences (a la ANSI C) are used to specify doublequotes, backslashes and control codes as string elements.

<code>\a</code>	Bell.
<code>\b</code>	Backspace.
<code>\t</code>	Tab.
<code>\n</code>	Newline.
<code>\v</code>	Vertical tab.
<code>\f</code>	Formfeed.
<code>\r</code>	Carriage return.
<code>\'</code>	Singlequote.
<code>\"</code>	Doublequote.
<code>\?</code>	Question mark.
<code>\\</code>	Backslash. Two consecutive backslash characters specifies a single backslash in the string.
<code>\ooo</code>	Code specified by the octal number <i>ooo</i> . Up to three octal digits are recognized.
<code>\xhh</code>	Code specified by the hexadecimal number <i>hh</i> . Arbitrarily many hexadecimal digits may be used.
<code>\newline</code>	The backslash character along with the newline code immediately following it are ignored. This sequence results in no characters at all in the string.

Example:

```
"The character \'\"' (doublequote)"
```

The above example is understood as a string containing the following characters.

```
The character '\"' (doublequote)
```

Strings should not contain newlines nor doublequotes directly. A standard way for including newlines within a string is to end the line with `'\n'`. By this, a new line code is inserted by the sequence `'\n'` and the actual newline in the source code following the second `'\n'` is ignored.

Unlike in Edinburgh Prolog, character strings are *not* lists of character codes.

### 3.3.4.2 Creating New Strings

In addition to the constant strings described above, strings can be dynamically created during execution. The following predicate can be used to create a new string.

```
new -String +Init +ElemSize [Object Creation on string]  
new_string -String +Init +ElemSize [Body Predicate on builtin]
```

A new string is created and unified with *String*. The last argument *ElemSize* specifies the bit width of the elements. As only 8-bit strings are available in the current version, this should be 8.

When the argument *Init* is an integer, it specifies the number of elements. In this case, the elements are initialized with integer 0 (null code). For example, `'generic:new(string, S, 3, 8)'` creates `"\0\0\0"`.

If *Init* is a list of integers, the newly created string is initiated by the elements of the list. Naturally, the number of elements of the string becomes the same as the length of the list. In this case, list elements should have values that fits in the given bit width; between 0 and 255 in case of 8-bit strings. For example, `'generic:new(string, S, [0'a, 0'b, 0'c], 8)'` creates `"abc"`.

### 3.3.4.3 Predicates on Strings

**string +String -Length -ElemSize** [Guard Method on **string**  
**string +String -Length -ElemSize** [Body Method on **string**  
**string +String -Length -ElemSize** [Guard Predicate on **builtin**  
 Tests whether *String* is a string object (if called in guard). The number of elements of *String* is returned in *Length* and the element size (which is always 8 in the current version) is returned in *ElemSize*.

**size +String -Length** [Body Method on **string**  
 Returns the number of elements of *String* in *Length*.

**element\_size +String -ElemSize** [Body Method on **string**  
 Returns the the element size of *String* in *ElemSize*.

**element +String +Index -Element** [Guard Method on **string**  
**element +String +Index -Element** [Body Method on **string**  
**string\_element +String +Index -Element** [Guard Predicate on **builtin**  
**string\_element +String +Index -Element** [Body Predicate on **builtin**  
 An element with index *Index* of the string *String* is unified with *Element*. The index is zero origin.

**less\_than +String1 +String2** [Guard Method on **string**  
**string\_less\_than +String1 +String2** [Guard Predicate on **builtin**  
 Succeeds only when *String1* is less than *String2* in lexicographical order.

**not\_less\_than +String1 +String2** [Guard Method on **string**  
**string\_not\_less\_than +String1 +String2** [Guard Predicate on **builtin**  
 Succeeds only when *String1* is not less than *String2* in lexicographical order.

**set\_element +Original +Index +Element -New** [Body Method on **string**  
**set\_string\_element +Original +Index +Element** [Body Predicate on **builtin**  
 -New

A new string is unified with *New*. The new string has the same elements as the *Original*, except that the *Index*'th element is updated to *Element*. The original string is left untouched. The index is zero origin.

**split +Original +At -Lower -Upper** [Body Method on **string**  
 The string *Original* is split at the index position *At* and the resultant two strings are unified with *Lower* and *Upper*. *At* has to be a non-negative integer less than or equal to the size of the original string. *Lower* will have elements with indices between 0 and *At*-1, inclusive. Elements with indices between *At* and up will be included in *Upper*.

**join** *+Lower +Upper -Joined* [Body Method on **string**]  
 Two strings *Lower* and *Upper* are concatenated together to make a new string *Joined*.

**search\_character** *+String +Start +End +Char* [Body Method on **string**]  
*-Where*

**search\_character** *+String +Start +End +Char* [Body Predicate on **builtin**]  
*-Where*

The character *Char* is searched for in *String*, beginning from the position *Start* and ending before *End*. If such a character is found, its index is unified with *Where*. If not, *Where* is unified with *-1*. The indices are zero origin.

In KLIC, creating a new string differing with only one element from the original is implemented with constant time and space overhead, regardless of the size of the string, by using multiversion array representation.

### 3.4 Handling Program Code as Data

KLIC allows higher order manipulation of executable code as data objects. Program modules are treated as *module* data objects and individual predicates are treated as *predicate* data objects.

#### 3.4.1 Modules

Program modules are treated as data through generic data objects of type **module**.

**new** *-Module +ModuleName* [Object Creation on **module**]  
 Creates a new object *Module* corresponding to the program module specified by *ModuleName* as a symbolic atom. If the specified module is not defined, the symbolic atom itself is returned to *Module*. See Section 2.6.1 [Creating Objects], page 8, for the format of object creation goals.

**module** *+Module* [Guard Method on **module**]  
 Tests whether *Module* is a module object or not.

**name** *+Module -ModuleName* [Body Method on **module**]  
 The module name of *Module* is returned to *ModuleName* as a symbolic atom.

#### 3.4.2 Predicates

Predicates in programs are treated as data through generic data objects of type **predicate**.

Predicate type data can be either denoted as a constant or created dynamically in runtime. Due to limitations of the features of host systems, dynamic creation may not be supported on some host systems.

The syntax of a predicate constant is as follows.

**predicate#**(*module:predicate/arity*)

Where *module* and *predicate* should be module and predicate name atoms and *arity* should be an integer (the number of arguments of the predicate). For example:

**predicate#**(main:main/0)    **predicate#**(quicksort:partition/4)

are valid predicate constants in programs.

Note that predicate constants are recognized by the KLIC compiler and not by the KLIC parser (see Section 3.6.2 [Input and Output with Prolog-like Interface], page 39). Thus, the notation described above means a usual data structure when simply read in using the Prolog-like I/O streams.

**new** *-Predicate +Module +PredName +Arity* [Object Creation on **predicate**]  
Creates a new object *Predicate* corresponding to the predicate specified by *Module* (a module object), *PredName* (a symbolic atom) and *Arity* (an integer). See Section 2.6.1 [Creating Objects], page 8, for the format of object creation goals.

**predicate** *+Predicate* [Guard Method on **predicate**]  
Tests whether *Predicate* is a predicate object or not.

**arity** *+Predicate -Arity* [Guard Method on **predicate**]  
**arity** *+Predicate -Arity* [Body Method on **predicate**]  
The arity of the predicate *Predicate* is returned to *Arity*.

**apply** *+Predicate +ArgVec* [Body Method on **predicate**]  
Calls the predicate specified by a predicate object *Predicate* with arguments specified by *ArgVec*. *ArgVec* should be a vector of arguments to be passed to *Predicate*. Thus, the size of the vector should match with the arity of the predicate.

**call** *+Predicate +Args...* [Body Method on **predicate**]  
Calls the predicate specified by a predicate object *Predicate* with arguments specified by *Args...*. The number of the arguments should match with the arity of the predicate.

**module** *+Predicate -Module* [Body Method on **predicate**]  
The program module that *Predicate* belongs to is returned to *Module* as a module data object.

**name** *+Predicate -Name* [Body Method on **predicate**]  
The name of the predicate *Predicate* is returned to *Name* as a symbolic atom.

## 3.5 Unix Interface

The module named **unix** makes features of the host operating system (Unix, typically) available to KL1 programs.

Almost all of the features are available as messages to a stream obtained by a predicate **unix/1** provided by the module **unix**. Some features are provided directly by predicates of the module.

### 3.5.1 Obtaining Unix Interface Stream

The module "unix" interfaces other programs through message streams. The stream can be obtained by calling the following predicate.

**unix** *?Stream* [Predicate on **unix**]  
A message stream corresponding to the Unix interface is returned to *Stream*.

Most of the features of the Unix interface are *not* provided as predicates, because no ordering is guaranteed between predicate calls.

If the Unix interface *were* provided as predicates, for example:

```
unix:cd("a", 0),
unix:cd("b", 0),
unix:system("mkdir ls", 0)
```

may list the directory `a` but may possibly list `b` or even some other directory before doing any `cd`, depending on the execution order. On the other hand:

```
unix:unix([cd("a", 0),
           cd("b", 0),
           system("ls", 0)])
```

will surely try two `cd` and `ls` in this order, as what decides the order is not the order of execution but the order of the elements in a list structure.

On parallel implementations, KLIC consists of multiple processes. The process in which the unix stream is obtained will be the process where all the messages are handled. For example, `cd(Path)` message will change working directory of that single process and none of others.

If you obtain two or more message streams, there will be no automatic synchronization between messages sent to different streams.

### 3.5.2 Opening Streams for Input and Output Operations

The following messages to the Unix stream open a Unix I/O stream. Messages to be sent to the resulting Unix I/O streams (*not* the Unix stream stream itself) for actually performing I/O are described in separate places: See Section 3.6.1 [Input and Output with C-like Interface], page 37, and Section 3.6.2 [Input and Output with Prolog-like Interface], page 39.

The following is a KLIC program for saying hello to the world.

```
main :- unix:unix([stdout(R)]), check_and_write(R).

check_and_write(normal(R)) :- R = [fwrite("hello world\n")].
```

<code>stdin</code>	<code>-Result</code>	[Message on unix stream]
<code>stdout</code>	<code>-Result</code>	[Message on unix stream]
<code>stderr</code>	<code>-Result</code>	[Message on unix stream]

These messages open a stream associated with process's standard input, standard output and standard error file respectively, and return `normal(Stream)` to *Result*.

<code>read_open</code>	<code>+Path -Result</code>	[Message on unix stream]
<code>write_open</code>	<code>+Path -Result</code>	[Message on unix stream]
<code>append_open</code>	<code>+Path -Result</code>	[Message on unix stream]
<code>update_open</code>	<code>+Path -Result</code>	[Message on unix stream]

These messages open the file named by the string *Path*, and return `normal(Stream)` to *Result*. The opening mode is input, output, append or input/output, respectively. If opening of the file fails, `abnormal` is returned instead.

### 3.5.3 Using Sockets

Unix- and Internet-protocol sockets can be obtained using the following messages to the Unix stream. Only sockets of SOCK\_STREAM type are provided.

**connect +Spec -Result** [Message on `unix stream`]

Creates a socket and connects it to socket specified by *Spec* and returns `normal(Stream)` to *Result*. *Spec* should have either of the following formats.

`unix(Path)`

A unix domain socket with the pathname *Path* is opened.

`inet(HostName, Port)`

An internet domain socket of the host specified by a string *HostName* and port number *Port* is opened.

`inet({B1, B2, B3, B4}, Port)`

An internet domain socket is opened. The host is specified by the internet address *B1* through *B4* is opened.

The obtained stream handles both input and output messages.

**bind +Spec -Result** [Message on `unix stream`]

Creates a socket and binds it to a name specified by *Spec*. The format of *Spec* is the same for the message `connect` except that *HostName* should be omitted for internet domain sockets. What is returned to *Result* is `normal(Stream)` but this *Stream* is a bound socket stream and does not directly handle I/O messages. Rather, it expects `accept` messages to obtain I/O message streams. When the bound socket stream obtained is closed and the socket type is `unix`, the named socket specified by *Path* in *Spec* will be unlinked.

**accept -Result** [Message on `bound socket`]

Accepts a connection to the socket and returns `normal(Stream)` to *Result*, where *Stream* is an I/O message stream for both input and output messages.

Sockets provide asynchronous I/O, that is, waiting for a connection or acceptance of a connection will not block other processes in the KLIC system. Trying to read or write to sockets with buffers empty or full respectively will not block the whole computation. Such I/O operations will be postponed until immediate operations become possible.

**Limitations:** When an operation on a socket is postponed, all the remaining operations to be done on the socket are also postponed until the completion of the postponed operation. This is problematic when both input and output has to be polled. The problem is planned to be solved in a future release.

**Limitations on Linux:** Asynchronous I/O operations do not work on Linux (at least with Slackware 1.2.0) with the current version.

### 3.5.4 Files and Directories

The following message to the unix stream handles files and directories.

**cd +Path -Result** [Message on `unix stream`]

Changes the working directory to *Path*. If successful, 0 is returned to *Result*; otherwise, -1 is returned. Corresponds to `chdir` system call.



**unlink** *+Path -Result* [Message on `unix stream`]  
 Removes the directory entry specified by *Path*. If successful, 0 is returned to *Result*; otherwise, -1 is returned. Corresponds to `unlink` system call.

**mktemp** *+Template -Filename* [Message on `unix stream`]  
 Makes a unique file name from the given *Template* and returns it to *Filename*. Corresponds to the C library routine `mktemp`. Unlike the library routine, the template does *not* have to have six trailing `X` characters. If a unique file name cannot be created somehow, a null string is returned to *Filename*.

**access** *+Path +Mode -Result* [Message on `unix stream`]  
 Checks accessibility of the file with pathname *Path* with the mode *Mode* is validate, and returns the result to *Result*. Corresponds to the C library routine `access`. If the file is accessible, 0 is returned; otherwise, -1 is returned. *Mode* is an integer, with the bits of the following meaning.

4	read permission
2	write permission
1	execute permission
0	test existence

**chmod** *+Path +Mode -Result* [Message on `unix stream`]  
 Changes the permission mode of the file with pathname *Path* to *Mode*. Corresponds to the system call `chmod`. If changing the mode is successful, 0 is returned to *Result*; otherwise, -1 is returned. *Mode* is an integer with standard Unix permission bits.

**umask** *-OldMask* [Message on `unix stream`]

**umask** *-OldMask +NewMask* [Message on `unix stream`]  
 Returns the current file creation mask to *OldMask*. With two arguments, sets the file creation mask to *NewMask*. Corresponds to the `umask` system call.

### 3.5.5 Handling Signal Interrupts

Unix signals can be converted to a list of integers using the following message to the `unix stream`.

**signal\_stream** *+Signal -Result* [Message on `unix stream`]  
 Unix signals specified by *Signal* (an integer value) will become caught and reported. The argument *Result* will become `normal(Stream)` and whenever a signal of the specified kind is detected, that signal number is sent to *Stream*. For example, if signal 2 (SIGINT in BSD and SVR4, at least) is detected, *Stream* becomes `[2|Rest]`. Further signals are reported to *Rest*.

**Limitations:** Signals may be ignored when they occur more than twice before the same kind of signal is detected, due to limitations of Unix.

### 3.5.6 Miscellaneous Messages to the Unix Stream

Various features of Unix are provided by sending the following messages to the unix stream.

**system** *+Command -Result* [Message on unix stream]

Executes *Command* (a string) in a newly created subshell, and returns its exit code to *Result*. Corresponds to the **system** system call.

**getenv** *+Name -Value* [Message on unix stream]

Returns the value of the environment variable with the name *Name* to *Value*. Corresponds to the library routine **getenv**. If such an environment variable does not exist, integer 0 is returned to *Value*.

**putenv** *+String -Result* [Message on unix stream]

The first argument *String* should be of form *Name = Value*. Adds or updates the environment variable *Name* with the value *Value*. Corresponds to the library routine **putenv**. If addition or updating is successful, 0 is returned to *Result*. Otherwise, non-zero integer value is returned.

**kill** *+Pid +Sig -Result* [Message on unix stream]

Sends the signal *Sig* to a process or a group of processes specified by *Pid*, and returns 0 on success or -1 on failure to *Result*.

**fork** *-Pid* [Message on unix stream]

Forks a new process which is a copy of the current process. Corresponds to the **fork** system call. If a child process is successfully created, the process ID of the child process is returned to *Pid* in the parent process, and 0 is returned in the child process.

**fork\_with\_pipes** *-Result* [Message on unix stream]

Creates pipes and fork a new process. The new process is a copy of the current process. In the parent process, *Result* is unified with **parent(Pid, In, Out)**, where *Pid* is the process ID of the newly created process. In the newly created child process, *Result* is unified with **child(In, Out)**. *In* and *Out* are Unix I/O streams to pipes; parent's *Out* is an output stream connected to child's *In*, which is an input stream; child's *Out* is connected to parent's *In*.

### 3.5.7 Predicate Interface

Some of the Unix interface are provided as predicates defined in the module **unix**.

**argc** *-Argc* [Predicate on unix]

Number of command line arguments not used by the KLIC system is returned to *Argc*. Such arguments start from the first argument not beginning with - or after -- in the command line.

**argv** *-ArgList* [Predicate on unix]

Command line arguments not used by the KLIC system is returned to *ArgList* as a list of strings.

**exit** *+ExitCode* [Predicate on unix]

Terminates the process immediately with the exit code *ExitCode*.

**times** *-Utime -Stime -CUtime -CStime* [Predicate on `unix`]

Returns process times in milliseconds. *Utime* is user time and *Stime* is system time. *CUtime* and *CStime* are those for children processes.

Note that when HZ (clock ticks per second) is not defined in some standard places, the system assumes 60.

## 3.6 Input and Output

KLIC provides two different sets of I/O operations. One is similar to those available from C language and the other is similar to those available from Prolog language.

C-like features are in a lower level and provide better performance both in speed and code size. However, during prototyping and debugging phases, the Prolog-like higher-level interface, allowing I/O of data structures directly, might be beneficial.

### 3.6.1 Input and Output with C-like Interface

Input and output operations with interface similar to those available in language C are described in this section.

Such interface are provided as messages to streams to open files, sockets, pipes &c, which are obtained by various messages to the Unix stream. See Section 3.5.2 [Opening Streams for Input and Output Operations], page 33.

#### 3.6.1.1 Common Messages with C-like Interface

The following messages are available for both input and output streams for C-like I/O.

**feof** *-Result* [Message on C-like I/O]

Returns 1 to *Result* if the stream is at the end of the file; otherwise 0. Corresponds to the library routine `feof`.

**fseek** *+Offset +Ptrname -Result* [Message on C-like I/O]

Changes the position of the stream according to the offset and pointer name given as *Offset* and *Ptrname*, respectively. The offset is specified as a signed integer by *Offset*. When *Ptrname* is 0, the offset is from the beginning of the file; when 1, from the current position; when 2, from the end of file. If successful, 0 is returned *Result*; otherwise -1.

Note that, due to the range restriction of integer values, this message may not be able to move to arbitrary positions in a very large file (larger than 128MB, on systems with 32-bit long integers).

**ftell** *-Result* [Message on C-like I/O]

Returns the offset of the current byte position to *Result*.

Note that, due to the range restriction of integer values, the obtained position may be incorrect for a very large file (larger than 128MB, on systems with 32-bit long integers).

**fclose** *-Result* [Message on C-like I/O]

Closes the stream. Returns 0 to *Result* if successful; -1 otherwise. No messages except for `sync/1` should be sent to a stream after closing.

**sync -Result** [Message on C-like I/O]  
 Returns 0 to *Result*. Useful in making sure that all the preceding messages have already been processed.

### 3.6.1.2 Input Messages with C-like Interface

The following messages are available for input streams for C-like I/O.

**getc -C** [Message on C-like I/O]  
 Reads one byte from the stream and returns it to *C*. At the end of file, -1 is returned.

**ungetc +C** [Message on C-like I/O]  
 Pushes back one byte *C* to the stream.

**fread +Max -String** [Message on C-like I/O]  
 Reads in at most *Max* bytes from the stream and returns the data as a byte string to *String*. Only up to 4,096 bytes can be handled in the current implementation. Note that the length of the resultant string may be smaller than the given maximum. This may happen at the end of the file for normal files and at any time for pipes or sockets.

**linecount -Count** [Message on C-like I/O]  
 Returns to *Count* the number of newline characters encountered so far. Within the first line of a file, it returns 0, as no newlines have been encountered yet. Conventional one-origin line numbers can be computed by adding one to this.  
 This line counting can be confused when **fseek/2** messages are used.

### 3.6.1.3 Output Messages with C-like Interface

The following messages are available for output streams for C-like I/O.

**putc +C** [Message on C-like I/O]  
 Writes one byte *C* to the stream.

**Number** [Message on C-like I/O]  
 Writes one byte *Number* to the stream. This is synonymous to **putc(Number)**.

**fwrite +String -Result** [Message on C-like I/O]  
 Writes out the contents of the byte string *String* to the stream and returns number of bytes actually written to *Result*. Note that the number of bytes actually written may be smaller than the length of *String*.

**fwrite +String** [Message on C-like I/O]  
 Writes out the contents of the byte string *String* to the stream. Unlike the **fwrite** message with *Result* argument, it waits for all the bytes in *String* to be output. This may be undesirable for streams that require unpredictable time period for output, such as internet sockets and pipes.

**fflush -Result** [Message on C-like I/O]  
 Flushes any output remaining on the stream. Returns 0 to *Result* if successful; -1 otherwise.

### 3.6.2 Input and Output with Prolog-like Interface

Unix interface streams with features to handle Prolog-like terms based on a operator precedence grammar can be obtained by the following predicate of module `klicio`.

The syntax of terms of KLIC is very close to that of Edinburgh Prolog but with subtle differences. See Section 3.2.1.1 [Notation of Atoms], page 17, Section 3.2.2.1 [Notation of Integers], page 19, Section 3.2.3.1 [Notation of Floats], page 21, Section 3.3.1.1 [Notation of Functors], page 23, Section 3.3.2.1 [Notation of Lists], page 25, Section 3.3.3.1 [Notation of Vectors], page 27, and Section 3.3.4.1 [Notation of Strings], page 28, for details.

#### 3.6.2.1 Opening Prolog-like I/O Streams

`klicio ?Stream` [Predicate on `klicio`]

A message stream corresponding to the Prolog-like term interface is returned to *Stream*. The obtained stream works the similar to a unix interface stream, which is used in turn to obtain message streams for actual I/O. I/O streams obtained through this stream accepts messages for Prolog-like term I/O described in this section in addition to ordinary C-like I/O messages.

This `klicio` stream is provided separately so that programs *without* the need of Prolog-like term I/O can be executable without modules for parsing and unparsing, as these modules have non-negligible sizes.

<code>stdin -Result</code>	[Message on <code>klicio</code> stream]
<code>stdout -Result</code>	[Message on <code>klicio</code> stream]
<code>stderr -Result</code>	[Message on <code>klicio</code> stream]
<code>read_open +Path -Result</code>	[Message on <code>klicio</code> stream]
<code>write_open +Path -Result</code>	[Message on <code>klicio</code> stream]
<code>append_open +Path -Result</code>	[Message on <code>klicio</code> stream]
<code>update_open +Path -Result</code>	[Message on <code>klicio</code> stream]

These messages open Prolog-like I/O streams. Messages to be sent to the resulting Prolog-like I/O streams (*not* the `klicio` stream itself) for actually performing I/O are described below.

These messages work exactly the same as the corresponding messages for `unix` streams, except that returned I/O streams understand messages for Prolog-like term I/O *in addition to* the messages for C-like I/O streams.

Prolog-like I/O streams are associated with operator definitions. Different operator definitions may be associated with each stream. Thus, adding or removing an operator to one stream will *not* affect operators used in other streams. Immediately after creation, each stream has a default set of operators.

#### 3.6.2.2 Common Messages with Prolog-like Interface

The following messages for C-like I/O streams can be used for Prolog-like I/O streams.

<code>feof -Result</code>	[Message on Prolog-like I/O]
<code>fseek +Offset +Ptrname -Result</code>	[Message on Prolog-like I/O]
<code>ftell -Result</code>	[Message on Prolog-like I/O]
<code>fclose -Result</code>	[Message on Prolog-like I/O]

- sync** *-Result* [Message on Prolog-like I/O]  
 See Section 3.6.1.1 [Common Messages with C-like Interface], page 37, for details.
- addop** *+Op +Type +Prec* [Message on Prolog-like I/O]  
 Adds an operator *Op* of type *Type* with precedence *Prec*.
- rmop** *+Op +Type* [Message on Prolog-like I/O]  
 Removes an operator *Op* of type *Type*.

### 3.6.2.3 Input Messages with Prolog-like Interface

- gett** *-Term* [Message on Prolog-like I/O]  
 Reads in a KLIC syntax term from the associated input stream to *Term*. On parsing errors, a message is output to **stderr** and another term is read in. At the end of the file, it returns an atom **end\_of\_file**.
- getwt** *-Result* [Message on Prolog-like I/O]  
 Reads in a KLIC syntax term from the associated input stream and returns the result to *Result*. *Result* will have the form **normal(WrappedTerm)** if parsing completes without errors. Here, *WrappedTerm* is a ground term representation of the term read in, where variables are represented as a ground term with the information on their names. On parsing errors, a message is output to **stderr** and another term is read in. At the end of file, it returns **normal(end\_of\_file)**.  
 See Section 3.6.2.5 [Wrapped Terms], page 41, for manipulation of wrapped terms.

The following messages for C-like I/O streams can also be used for Prolog-like I/O streams.

- getc** *-C* [Message on Prolog-like I/O]  
**ungetc** *+C* [Message on Prolog-like I/O]  
**fread** *+Max -String* [Message on Prolog-like I/O]  
**linecount** *-Count* [Message on Prolog-like I/O]  
 See Section 3.6.1.2 [Input Msgs (C style)], page 38, for details.

### 3.6.2.4 Output Messages with Prolog-like Interface

- putt** *+Term* [Message on Prolog-like I/O]  
**puttq** *+Term* [Message on Prolog-like I/O]  
**putwt** *+WrappedTerm* [Message on Prolog-like I/O]  
**putwtq** *+WrappedTerm* [Message on Prolog-like I/O]  
 A term *Term* or a wrapped term *WrappedTerm* is written out to the associated output stream.

Messages *without* the character **q** are supposed to omit two quotes around symbolic atoms even when they are required to be correctly read in again. However, currently they work exactly the same as the messages with **q**.

With the current version, the output format is meant only to be machine-readable and not so readable for humans. That is, no operators are used and all atoms are enclosed within parentheses.

See Section 3.6.2.5 [Wrapped Terms], page 41, for manipulation of wrapped terms.

The following messages for C-like I/O streams can also be used for Prolog-like I/O streams.

<code>putc +C</code>	[Message on Prolog-like I/O]
<code>Number</code>	[Message on Prolog-like I/O]
<code>fwrite +String -Result</code>	[Message on Prolog-like I/O]
<code>fwrite +String</code>	[Message on Prolog-like I/O]
<code>fflush -Result</code>	[Message on Prolog-like I/O]

See Section 3.6.1.3 [Output Messages with C-like Interface], page 38, for details.

Note that a period to end a term is not written out by these messages. Thus, writing out a period and a space or a newline character is usually required for the output to be read in again. The following goal sequence opens the file named `/tmp/foo.bar`, waits full instantiation of the variable `X`, and then outputs the value in a Prolog-like format followed by a period and a newline.

```
klicio:klicio([write_open("/tmp/foo.bar", normal(S))]),
S = [putt(X), putc(0'.), nl].
```

<code>nl</code>	[Message on Prolog-like I/O]
-----------------	------------------------------

Outputs a newline code. This is synonymous to sending a message `putc(10)` to the same stream.

Note that Prolog-like I/O streams also accept all the messages accepted by C-like I/O such as `putc/1` or `getc/1` (see Section 3.6.1 [Input and Output with C-like Interface], page 37).

### 3.6.2.5 Wrapped Terms

To allow metalevel manipulation of terms including variables, KLIC provides a data representation called *wrapped term*. A wrapped term is a ground term without any variables in it. A wrapped term has one of the following forms.

<code>variable(VarName)</code>	a variable with its name string <i>VarName</i>
<code>atom(Atom)</code>	a symbolic atom <i>Atom</i>
<code>integer(Int)</code>	an integer <i>Int</i>
<code>floating_point(Float)</code>	a floating point number <i>Float</i>
<code>list([Car Cdr])</code>	a cons cell consisting of <i>Car</i> and <i>Cdr</i> , which are wrapped terms recursively
<code>functor(Functor(Arg, ...))</code>	a functor structure; its arguments ( <i>Arg</i> , etc) are wrapped terms recursively
<code>vector({Elem, ...})</code>	a vector; its elements ( <i>Elem</i> , etc) are wrapped terms recursively

**string**(*Str*)

a string *Str*

**unknown**(*Term*)

some unknown data; wrapping may be imprecise in this case

For example, the wrapped representation of a term:

`f(a, X, {3, ["abc"|X]}, 3.14)`

is the following.

```
functor(f(atom(a),
           variable("X"),
           vector({integer(3), list([string("abc")|variable("X")])}),
           floating_point(3.14))).
```

The following predicate convert wrapped terms to normal terms.

**unwrap** -*Wrapped* ?*Term* [Predicate on *variable*]  
 Converts a wrapped term *Wrapped* to a normal term *Term*.

Wrapped terms are normally obtained as a result of input operations (see Section 3.6.2.3 [Input Messages with Prolog-like Interface], page 40). Wrapped terms can also be constructed by usual user programs, as they are nothing more than a usual KL1 term. The following predicate that converts normal terms to wrapped terms may also be useful in certain cases.

**wrap** ?*Term* -*Wrapped* [Predicate on *variable*]  
 Converts a normal term *Term* to a wrapped term *Wrapped*.

In the current version, all variables are given the same name `_`. Thus, by wrapping a term and then unwrapping its result, all the variables in the original term will become references to the same variable. This is a bug and is planned to be fixed in a future version.

When *Term* contains multiple references of a same variable, computation on-going concurrently may instantiate the variable. In such cases, this predicate may yield a wrapped term in which two original occurrences of the same variable are converted differently; one as a variable and another as a non-variable term. This is an inherent problem of the specification of this predicate and probably will never be fixed. Thus, applying this predicate to non-ground terms should be restricted to certain metalevel programs, such as debugging utilities.

### 3.7 Controlling System Behavior

The following predicates are provided in the module `system_control`.

**postmortem** +*Module* +*Goal* -*Result* [Predicate on `system_control`]

Registers postmortem processing after normal or abnormal termination of the main program. *Goal* should be a functor structure specifying the predicate and arguments of the postmortem processing goal. *Module* should be a symbolic atom specifying the module of the postmortem processing predicate. Only a single goal can be specified; comma-separated sequences of goals are not allowed.



When the registration is done, *Result* is unified with []. Waiting for this will prevent further processing to be executed before the completion of the registration.

If this predicate is called many times, the last registration will be effective.

**gc -Before -After** [Predicate on `system_control`]  
 Requests garbage collection and returns the heap size in words before and after the garbage collection to *Before* and *After* respectively. The size of a word is the same as the size of type `long` of the C language system used in the installation.  
 In parallel implementations, only garbage collection for local storage is requested. Requesting of global garbage collection is not available.

### 3.8 Timer

KLIC provides real-time timers. Although Unix provides only one timer per process, KLIC virtualizes the mechanism and provides as many timers as needed.

Implementations on host systems where real-time timers are not available do not provide the feature.

Time values (both times and time intervals) are represented by a term of the form `time(Day, Sec, Usec)`, where *Day*, *Sec* and *Usec* are non-negative integers representing days, seconds and microseconds. *Sec* should be less than 86,000 (one day) and *Usec* should be less than 1,000,000 (one second).

The following predicates are provided in the module `timer`.

**get\_time\_of\_day -Time** [Predicate on `timer`]  
 The current time expressed in seconds and microseconds since midnight of January 1, 1970 GMT is returned to *Time*.

The time obtained is that of when a goal of this predicate is actually executed. Note that ordering of goal execution is up to the KLIC system. The time reported is only guaranteed to be the time between two observable events: when the parent goal of this goal is reduced, and when the value of *Time* is inspected.

Note also that the reported time is what is returned by the underlying operating system of the worker task. On a distributed system, clocks of constituting systems may not agree completely.

**add Time1 Time2 -Time** [Predicate on `timer`]  
**sub Time1 Time2 -Time** [Predicate on `timer`]

Computes sum and difference of two time values, respectively.

**compare Time1 Time2 -Result** [Predicate on `timer`]  
 Compares two time values *Time1* and *Time2* and returns the result in *Result*. The result is `<` if *Time1* is smaller than (or before) *Time2*, `=` if they are the same, `>` if *Time1* is larger than (or after) *Time2*.

**instantiate\_at Time -Var** [Predicate on `timer`]  
**instantiate\_after Interval -Var** [Predicate on `timer`]

Unifies *Var* with a symbolic atom [] at the time specified. The former predicate does this *at* the specified time, while the latter does this *after* the specified time interval. If the specified time has already passed, the variable may be instantiated immediately.

Note that instantiation may be delayed arbitrarily long. Reasonable implementations should have short delays.

**instantiate\_every** *Interval Stop -Var* [Predicate on **timer**]

Incrementally instantiate *Var* with a list of symbolic atom []. The first element is instantiated after the time interval specified, the second after time twice the specified value, etc. It will be repeated forever unless the argument *Stop* becomes instantiated, on that occasion, the list will be terminated.

Note that instantiation may be delayed arbitrarily long. Reasonable implementations should have short delays.

### 3.9 Random Number Generator

Pseudo-random numbers can be generated using the object class **random\_numbers**. This random number generator is based on **nrnd48**. The random number generator feature is not available if **nrnd48** is not on the host system.

**new** -*Randoms Range* [Object Creation on **random\_numbers**]

**new** -*Randoms Range Seed* [Object Creation on **random\_numbers**]

An infinitely long list of pseudo-random integers ranging between 0 and *Range* - 1, inclusive, is returned to *Randoms*. *Range* should be a positive integer. The optional argument *Seed* specifies the seed for random number generation. The list elements are guaranteed to be the same if the same seed is given.

Note that, although the list is virtually infinite, elements are computed lazily on demand as programs incrementally inspect their values.

## 4 Using KLIC

This chapter describes how to use the KLIC system.

### 4.1 Compiling Programs with KLIC

After proper installation, KL1 programs can be compiled into C program and then to executables by the command `klic`. `klic` is a compiler driver that allows various options.

#### 4.1.1 Command for Compilation

By simply running `klic` command with the name of KL1 program source file with the trailing `.kl1` as an argument, that program will be compiled into C and then to an executable format.

For example, to compile `XXX.kl1`, type in:

```
% klic XXX.kl1
```

The compilation result will be found in `a.out`. If you want the compilation result to be named `YYY`, do the following.

```
% klic -o YYY XXX.kl1
```

If your program is divided into several files, say `XXX.kl1`, `YYY.kl1` and `ZZZ.kl1`, you can compile and link them together by the following.

```
% klic XXX.kl1 YYY.kl1 ZZZ.kl1
```

It is also possible to separately compile several KL1 source files and link them afterwards. To avoid linkage errors, you have to stop before linkage by giving the `-c` flag, as follows.

```
% klic -c XXX.kl1
```

```
% klic -c YYY.kl1
```

```
% klic -c ZZZ.kl1
```

Finally, you have to link all of them together by the following.

```
% klic XXX.o YYY.o ZZZ.o
```

See See Section 4.1.2 [Compiler Options], page 45, for details of compilation flags.

If you want to link program pieces written directly in C, say `CCC.c` and `DDD.c`, with pieces written in KL1, `XXX.kl1` and `YYY.kl1`, simply do the following.

```
% klic CCC.c DDD.c XXX.kl1 YYY.kl1
```

The order of files specified does not matter. C functions can be invoked from within inline-expanded codes (see Section 2.10 [Inline C Code], page 12).

#### 4.1.2 Compiler Options

Options available for the compilation command `klic` are listed below.

- `-c`            Stop after generating relocatable object and don't link the program.
- `-C`           Stop after translation into C.
- `-d`           Don't try any compilation (dry run). Implies `-v`.
- `-D database_manager`  
              Use the specified database manager program.

- g**            Debug flag passed to the C compiler.
- I *directory***  
              Use the additional include directory specified for C compilation.
- K *klic\_compiler***  
              Use the specified KL1 to C translator program.
- l *library***  
              Use the additional library specified for linking.
- L *directory***  
              Use the additional directory specified to be searched for **-l**.
- o *file***    Use the file name for the generated executable file.
- O**
- O*level***    Use the specified optimization level. When a non-zero optimization level is specified, some additional optimization flags may be also passed to the C compiler. Such Additional optimization flags are system dependent and determined on KLIC system installation procedure.  
              For this option, no spaces are allowed between **-O** and *level*.
- P *parallel***  
              Run subtasks (C compilers &c) in parallel. At most *parallel* subtasks are forked at a time.
- R**            Do recompilation regardless of file dates.
- S**            Stop after generating assembly code output.
- n**            Link with the non-debugging version of the runtime library. By default, the debugging version is used.
- v**            Run in verbose mode. All the commands executed through the compiler driver will be output to standard error.
- x*directory***  
              Use database file **klic.db** in the specified directory and also place **atom.c**, **funct.c** and **predicates.c** and their corresponding objects in the same directory. This flag is useful when programs to be linked together are distributed to multiple directories.
- X*directory***  
              Initiate the database file **klic.db** from the database initiation file **klicdb.init** under the specified directory, when the database file does not exist yet. It defaults to the default library directory.

The following environment variables can change the default behavior of the compiler. Options given at compilation time supersede the environment variable values.

#### KLIC\_LIBRARY

Directory for runtime libraries. Superseded by the **-X** option.

#### KLIC\_DBINIT

Directory for initial database. Defaults to the directory for runtime libraries.

**KLIC\_COMPILER**

KL1 to C translator program. Superseded by the `-K` option.

**KLIC\_DBMAKER**

Database manager program. Superseded by the `-D` option.

**KLIC\_INCLUDE**

Additional include directory for C compilation. Superseded by the `-I` option.

**KLIC\_CC** C compiler to be used.

**KLIC\_CC\_OPTIONS**

Additional option flags for the C compiler.

**KLIC\_LD** Linker to be used.

**KLIC\_LD\_OPTIONS**

Additional option flags for the linker.

### 4.1.3 How KLIC Compiler Works

Understanding how KL1 programs are compiled and executed may help understanding the usage of KLIC in further depth.

The system consists of the following three modules.

- KLIC compiler
- KLIC database manager
- KLIC runtime system

KL1 programs are compiled using the KLIC compiler into C programs. It also generates files *FILE.ext* containing information on atoms and functors used in the program. The information in *.ext* files for programs to be linked together is merged together later by the database manager, into files *atom.h*, *funct.h*, *atom.c*, *funct.c* and *predicates.c*.

The object C program is then compiled by a C compiler, with headers provided by the KLIC runtime system, *atom.h* and *funct.h*. The files *atom.c*, *funct.c* and *predicates.c* are also compiled, and linked together with the runtime system (*predicates.c* is linked only with debugging runtime).

Compilation, database management and linkage are governed by a driver program named *klic*. This program *klic* plays a role similar to *cc* and *make* combined. *cc* controls the C preprocessor, the C compiler kernel and the linker; *klic* controls the KL1-to-C compiler, the C compiler, the KL1 program database manager and the linker. *make* selectively executes compilation only when needed by examining the file dates; *klic* works similarly.

## 4.2 Running Programs Compiled with KLIC

You can simply run the compiled executable. If you compiled your program into the file *a.out*, you simply give the file name *./a.out* to the shell you are using.

The predicate *main* with no arguments in the module *main* will be the initial goal to be executed (see Section 2.5 [Initial Goal], page 7).

### 4.2.1 Runtime Switches for Programs Compiled with KLIC

The following options are available on running the compiled executable.

- h size** Specifies initial heap size in words. As copying garbage collection is used, memory size actually used for heap will be twice this size. The size can be specified directly (such as 2097152) or with a postfix **k** or **m** (as 2048k or 2m) to specify units of  $2^{10}$  or  $2^{20}$  words. The default heap size is determined by the macro **HEAPSIZE**, which is 24k in the original distribution. The length of one word is the same as the length of the type **long int** in C, that depends on the hardware and the C compiler you use. The heap size will be increased automatically according to options **-H** and **-a**.
- H size** Specifies maximum heap size in words. Automatic heap expansion mechanism will never try to expand the heap above the size specified by this option. The default value is infinite.
- a ratio** Specifies threshold active cell ratio as a floating point number. If the ratio of the space occupied by active (non-garbage) cells in the heap space is above this threshold, the heap size will be doubled in the next garbage collection, as far as the size doesn't exceed the maximum size specified by the **-H** option. The default value is 0.5.
- g** Specifies that time required for garbage collection is to be measured. As garbage collection will not take long for small heap sizes, the measurement overhead can be more than that. Thus, by default, garbage collection timing is disabled.
- s** Specifies suspension statistics. After execution of the program, suspended predicates and numbers of their suspensions are reported. This option is available when the debugging version of the runtime library is linked, which is the default setting (see Section 4.1.2 [Compiler Options], page 45).
- t** Specifies to start execution with tracing (see Section 4.3 [Tracing Program Execution], page 48). Tracing is only possible when the debugging version of the runtime library is linked, which is the default setting. The non-debugging version of the runtime library can be specified by compilation time options (see Section 4.1.2 [Compiler Options], page 45).

When all the ready goals have been executed, the program will stop. If there remain any goals awaiting for input data and if the program is linked with the debugging runtime library, it will try to detect which goal is problematic and report such a goal. Otherwise, if the linked library is a non-debugging version, only the number of such remaining goals is reported.

## 4.3 Tracing Program Execution

KLIC provides a debugging tracer with *spying* (break point) feature.

### 4.3.1 Preparation for Traced Execution

To use the tracing feature, you have to link your program with the debugging version of the runtime library. The debugging version is used by default, but when you give the **-n** option to the compilation command **klc**, tracing will not be available.

If you already have compiled and linked the program with the `-n` option, you don't have to recompile the program from scratch; running the command `klc` again without the `-n` option will only link the object with the debugging version of the runtime library, which takes much shorter time.

To trace execution of a program, simply run your program with `-t` option (see Section 4.2.1 [Runtime Switches for Programs Compiled with KLIC], page 48).

### 4.3.2 Trace Ports

Execution of KL1 programs proceeds as follows.

1. The initial goal `main:main` is put into a pool of goals to be executed.
2. One goal is taken from the goal pool (`CALL`).
3. The goal is matched against the program clauses.
4. If any of the clauses matches the goal, the goal is reduced into subgoals and they are put back to the goal pool (`REDUCE`).
5. If no clause matches the goal, then the whole computation will be aborted (`FAIL`).
6. If values of goal arguments or their substructures are not defined and thus it is not possible yet to decide whether some clauses will match the goal or not, the goal is put into another goal pool awaiting for required values (`SUSPEND`).
7. If there still remain some goals in the goal pool, loop back to the step 2.

Execution of a goal can be traced on four of the above listed points, numbered 2, 4, 5 and 6. Such points of interest are called *trace ports* and referenced as `CALL`, `REDUCE`, `FAIL` and `SUSPEND` ports, respectively.

Those who are accustomed to the four-port trace model of Prolog may wonder why two other ports of Prolog, `EXIT` and `REDO` are missing. The `REDO` port does not exist because KL1 programs do not backtrack. The `EXIT` port is not traced for two reasons. First, keeping track of all the goal-subgoal hierarchy is much more costly for a concurrent language such as KL1 than for sequential languages such as Prolog. Many different subtrees of the hierarchy may run interleaving each other, because of the data-flow synchronization feature. The other reason is that, KL1 programs are often written as a set of communicating processes each defined as a goal calling the same predicate in a tail-recursive fashion. Such processes (sometimes called *perpetual processes*) will almost never finish and detecting their termination is not as meaningful as in Prolog.

### 4.3.3 Format of Trace Display

Below is our sample program for explanation here.

```
:- module main.

main :- nrev([1,2],X), builtin:print(X).

nrev([], R) :- R = [].
nrev([W|X], R) :- nrev(X, XR), append(XR, [W], R).

append([], Y, Z) :- Z = Y.
append([W|X], Y, WZ) :- WZ = [W|Z], append(X, Y, Z).
```

Listed below is output of a full trace of execution of the sample program.

```

1 CALL:main:main?
1 REDU:main:main :-
2   0:+nrev([1,2],_4)
3   1:+builtin:print(_4)?
2 CALL:main:nrev([1,2],_4)?
2 REDU:main:nrev([1,2],_4) :-
4   0:+nrev([2],_D)
5   1:+append(_D,[1],_4)?
4 CALL:main:nrev([2],_D)?
4 REDU:main:nrev([2],_D) :-
6   0:+nrev([],_18)
7   1:+append(_18,[2],_D)?
6 CALL:main:nrev([],_18)?
6 REDU:main:nrev([],[])?
7 CALL:main:append([],[2],_D)?
7 REDU:main:append([],[2],[2])?
5 CALL:main:append([2],[1],_4)?
5 REDU:main:append([2],[1],[2|_1F]) :-
8   0:+append([],[1],_1F)?
8 CALL:main:append([],[1],_1F)?
8 REDU:main:append([],[1],[1])?
3 CALL:builtin:print([2,1])?
[2,1]
3 REDU:builtin:print([2,1])?

```

As this program does not make any suspensions nor failures, all the trace outputs here are either at the call or the reduce port (marked as REDU).

The first line of the above is the trace of the call port of the initial goal `main:main`.

```
1 CALL:main:main?
```

All the traced goals are given a unique identifier (an integer value) to distinguish them among themselves. The number 1 in the first column here is the identifier of the initial goal.

The initial goal matches the first clause defined in the program and thus reduced into subgoals as defined in the program clause. This reduction is traced as follows.

```

1 REDU:main:main :-
2   0:+nrev([1,2],_4)
3   1:+builtin:print(_4)?

```

This shows that the original goal `main:main` with ID 1 has been reduced into two new goals, `main:nrev([1, 2], _4)` and `builtin:print(_4)`, with IDs 2 and 3 respectively.

The numbers 0 and 1 following the IDs 2 and 3 of the two new goals are sequential numbers for the subgoals generated by the reduction. They are used by some tracer commands to identify which subgoal to apply the command to. Unlike unique goal IDs that have global meaning, these subgoal numbers are meaningful only at this specific port.

Next comes `:`, which means the subgoal is an ordinary subgoal of the parent goal. There are other possibilities here. The character `*` means that the goal following it is also a



subgoal, but is given a priority different from the parent. The priority is displayed in a pseudo-pragma format. The character `!` means that the goal following it is not actually a subgoal reduced from the parent goal, but is a goal awaiting for some variable value which has just waken up as this reduction gave some concrete value to the variable. The character `#` similarly indicates a goal waken up, but with a priority different from the parent.

Then comes either `+` or `-`. `+` means that the subgoal will be traced if you simply continue the execution, and `-` means it will not. This can be changed by giving some tracer commands described below. In the example above, all of the subgoals have `+` as all goals are traced.

Then the module name, a colon character, and the predicate name of the subgoal are displayed. The module name for predicates defined in the same module as the predicate of the parent goal is omitted with the colon for brevity. In the above example, the subgoal calling `nrev` (that is `main`) does not have its module name displayed, as it is the same as the parent goal `main:main`.

Finally comes the argument list in parentheses separated by commas. The second argument of `nrev` and the only argument of `print` is `_4`, which corresponds to a variable corresponding to `X` in the source program. As variables are newly allocated for all incarnation of predicate clauses, and as two or more variables can be unified together, displaying their original names in the source program is not meaningful. They are given unique names such as `_4`.

Actually, this number 4 is related to the physical memory address of the variable. It will thus change completely by garbage collections. However, as garbage collections are not so frequent, the address information is still quite useful for debugging.

The trace output stops after displaying all the subgoals and a question mark. Here, you can input one of the trace commands described below.

#### 4.3.4 Trace Controlling Commands

Tracing can be controlled at each *leashed* port (see Section 4.3.6 [Controlling Trace Ports], page 53). Tracing can be controlled for the traced goal as a whole or, at the reduce port, for each of the newly created subgoal. The default of whether or not to trace goals of each predicate can also be set.

##### 4.3.4.1 Controlling Tracing of the Traced Goal

The following commands are available for controlling program execution.

Continue: `c` or simply `RET`

Continues stepping execution. Subgoal marked as `-` are not traced even in stepping mode.

Leap: `l` Continues execution without tracing until a spy point is encountered. See Section 4.3.5 [Spying], page 53, for details.

Skip: `s` Continues execution of the traced goal and all subgoals thereof without tracing them at all. Even spy points are neglected.

Abort: `a` Aborts whole execution of the program.

These commands do not take any arguments.

#### 4.3.4.2 Controlling Tracing of Newly Created Subgoals

Tracing of each subgoal (displayed as + or -) can be changed by the following commands.

Trace: + *subgoal\_number* . . .

Switches on the trace of the specified subgoal(s). Multiple subgoal numbers separated by spaces can be specified. If no subgoal numbers are given, all the subgoals become traced.

No Trace: - *subgoal\_number* . . .

Switches off the trace of the specified subgoal(s). Multiple subgoal numbers separated by spaces can be specified. If no subgoal numbers are given, all the subgoals become untraced.

Toggle Trace: *subgoal\_number* . . .

Toggles the trace switch of the specified subgoal(s). Multiple subgoal numbers separated by spaces can be specified.

#### 4.3.4.3 Changing Default Trace of Predicates

By default, all the subgoals of a goal will have trace switch on (+) initially at the reduce port. This default setting can be changed predicate by predicate using commands described in this section, so that predicates you are not interested in will not be traced by default.

In what follows, command arguments <predicate> has one of the following format.

*Module:Predicate/Arity*

Specifies explicitly and exactly one predicate. For example, *main:nrev/2*.

*Module:Predicate*

Specifies all the predicates within a module with different arities.

*Module:* Specifies all the predicates defined in a module. Note that a colon is required after the module name to distinguish it from a predicate name.

*Predicate/Arity*

Specifies the predicate defined in the same module as the predicate of the currently traced goal with the given name and arity.

*Predicate*

Specifies all the predicates defined in the same module as the predicate of the currently traced goal with the given name.

Listed below are commands to change the default for given predicates.

No Trace Default: n *Predicate* . . .

Sets the default trace for the predicate(s) to be off. If no predicates are given as argument, the predicate of the traced goal is considered to be specified.

Trace Default: t *Predicate* . . .

Sets the default trace for the predicate(s) to be on. If no predicates are given as argument, the predicate of the traced goal is considered to be specified.

### 4.3.5 Spying

It is often the case that only some specific predicates are of interest for debugging. In such cases, ports for such predicates can be specified as the *spy points*. You can let program run without tracing until some spy point is encountered, using the `leap (1)` command. See Section 4.3.4.1 [Controlling Tracing of the Traced Goal], page 51, for details.

Commands described in this section set or reset such spy points.

Spy: `S Predicate ...`

Makes the predicate(s) spied. If no predicates are given as argument, the predicate of the traced goal is spied.

No Spy: `N Predicate ...`

Resets the spy point on the predicate(s). If no predicates are given as argument, the spy point on the predicate of the traced goal is reset.

### 4.3.6 Controlling Trace Ports

The four trace ports can be selectively enabled and disabled. Disabled ports will not be traced at all.

In addition, for each port, you can specify whether to stop and wait for command input. Ports where execution stops and waits for commands are said to be *leashed*. On ports enabled but not leashed, the trace output will be displayed but execution continues as if the `continue` command (carriage return) was input immediately. For spied predicates, even unleashed ports will be leashed.

Commands described in this section is for controlling such attributes of ports. They take port names as their arguments, specified as one of the following ways.

Call: `c, call`

Reduce: `r, redu, reduce`

Suspend: `s, susp, suspend`

Fail: `f, fail`

All ports: `a, all`

Listed below are the commands for controlling ports.

Enable Port: `E port ...`

Enables the specified port(s).

Disable Port: `D port ...`

Disables the specified port(s).

Leash Port: `L port ...`

Leashes the specified port(s).

Unleash Port: `U port ...`

Unleashes the specified port(s).

### 4.3.7 Display Control Commands

Sometimes, full information of the traced goals is not desirable, as too much information is only harmful for understanding the program behavior. Thus, commands in this section are provided for controlling the amount of information displayed on trace ports.

The amount of display is controlled by a combination of the following options.

- By limiting display depth: Arguments of structures below depth limit are displayed in the following abbreviated way.

$$\begin{array}{ll} f(a,b,c,d,e) & \mapsto f(\dots) \\ [a,b,c,d,e] & \mapsto [\dots] \end{array}$$

- By limiting display length: Argument lists of structures or character strings longer than the length limit are displayed in the following abbreviated way.

$$\begin{array}{ll} f(a,b,c,d,e) & \mapsto f(a,b,c,\dots) \\ [a,b,c,d,e] & \mapsto [a,b,c,\dots] \\ \text{"abcde"} & \mapsto \text{"abc\dots"} \end{array}$$

- By specifying a subterm to be displayed: Only a part of the traced goal can be specified for display.

The following commands can be used to control the options.

Set Print Depth: `pd depth`

Sets depth limit of displaying data structures to *depth*. With no argument, prints the current depth limit value.

Set Print Length: `pl length`

Sets length limit of displaying data structures to *length*. With no argument, prints the current length limit value.

Toggle Verbose Print: `pv`

Toggles verbose printing mode switch. In verbose printing mode, variables with goals awaiting for its value are displayed with the information of the goal.

Set Subterm: `^ N`

Reset Subterm: `^`

Sets the *N*-th subterm of the traced goal to be inspected. With 0 specified as *N*, the subterm goes up one level. With *N* omitted, subterm inspection is reset. For list structures, 1 means car and 2 means cdr.

With subterm specification, only the subterm of the traced goal is displayed after the information of which subterm is inspected. An example follows.

```
10 CALL: foo:bar(f(a,g(...),[...]))? ^1
10 CALL: ^1 f(a,g(b,c),[d,e])? ^2
10 CALL: ^1^2 g(b,c)? ^0
10 CALL: ^1 f(a,g(b,c),[d,e])? ^3
10 CALL: ^1^3 [d,e]? ^2
10 CALL: ^1^3^2 [e]? ^
10 CALL: foo:bar(f(a,g(...),[...]))?
```

At reduce ports, subgoals created by the reduction are not displayed when subterm display is specified; only the specified subterm of the parent goal is

displayed. With the current version, vector elements cannot be specified as subterms.

The initial setting of depth and length limits are 3 and 7, respectively. Verbose print mode is initially switched off.

### 4.3.8 Dumping Goals

It is desirable sometimes to dump all the goals in the system as a last resort. The following commands do it.

Dump Ready Queue: `Q`

Displays all the goals in the ready queue (goal pool) with their priorities.

Dump Suspended (Waiting) Goals: `W`

Displays all the suspended goals in the system with their priorities.

### 4.3.9 Miscellaneous Commands

Status Query: `=`

Displays tracer status information, such as follows.

```

      port: Call Susp Redu Fail
enabled:  +    +    +    +
leashed:  +    +    +    +
print terse; depth = 3; length = 7

```

List Modules: `lm`

Lists all the modules of the currently executed program.

List Predicates: `lp`

Lists all the predicates and their default trace status of the currently executed program.

Queue: `Q` Lists the contents of the ready queue (goal pool).

Help: `? or h`

Lists all the commands and their terse description available at the current port.

### 4.3.10 Detecting Perpetual Suspensions

When some goals are awaiting for instantiation of a variable that will never be instantiated by any other goals, such goals will never proceed. This situation is called *perpetual suspension*. Perpetual suspension is detected by the garbage collector of KLIC. Thus, during program execution, garbage collections may find perpetual suspensions.

The system keeps track of the number of suspended goals. When there exist no goals ready for running and there are suspended goals remaining, the system will try garbage collection to detect perpetual suspensions.

Perpetual suspensions are reported as follows.

```

!!! Perpetual Suspension Detected !!!
3 PSUS: Module:Predicate(Args...)?

```

The same command set as at a fail port is available here.

## 4.4 Installation

Installation of KLIC should be fairly easy.

Host-dependent and preference-based customizations are made by running a configuration script provided with the distribution. Then `make all` should compile the whole system. You can make sure that the system has been compiled without problems by running `make tests`. Then you can install the system by `make install`.

### 4.4.1 Configuration

First thing to do in installation of KLIC is to configure the KLIC system depending on the host computer system and your preference.

Go to the root directory of the distribution (referred to as *ROOT* in what follows). Then, run the configuration script there by a command `./Configure`. The script will search for available software tools in your system and ask your preferences.

The default shell programs on some Unix systems based on BSD 4.2 do not understand some of the constructs used in this configuration script. In such a case, obtain a modern shell (such as GNU `bash`) and let it execute the script, as follows.

```
% bash Configure
```

If you have built the system before and rebuilding it in the same directory, it will ask whether the same values you specified the last time should be used as default values.

The next question it asks (or the first, if it is the first time to build the system) is whether to configure also for parallel implementations. If you want to install only the sequential system, please answer `no` to the question. See relevant sections (see Section 4.5 [Distributed KLIC], page 57, and see Section 4.6 [Shared-Memory KLIC], page 59), for further details of configuraion of parallel versions of the system.

The configuration script will make three files.

```
ROOT/Makefile
ROOT/include/klic/config.h
ROOT/config.sh
```

The last one records the specified options for reconfiguration.

The configuration script asks about the parallelism used in the installation procedure. You can specify non-zero parallelism here to speed up the procedure if you are installing your system on a lightly loaded multiprocessor system. Do *not* use parallel execution features of the `make` program.

### 4.4.2 Compiling the KLIC system

After configuring the system, typing in `make all` should compile the whole KLIC system, including the KL1 to C compiler and the runtime libraries.

### 4.4.3 Testing the Compilation

After system compilation is finished, you are recommended to test whether the compilation went without problems. To do that, type in `make tests` in the root directory of the distribution (not in its subdirectory `test`). This will compile and run several KL1 test programs and compares the output with the expected output.

#### 4.4.4 Installing the Objects

After compilation, typing in `make install` will install the compiler, header files and runtime libraries to directories specified on configuration (see Section 4.4.1 [Configuration], page 56).

#### 4.4.5 Cleaning Up the Installation Directory

After installation has been done, typing in `make distclean` will delete all the files *not* included in the distribution.

Normal users should **not** try `make realclean`, which will delete C program source files generated from KL1. A working KL1 to C compiler will be needed to regenerate the C program source files.

#### 4.4.6 When Something Goes Wrong

When the installation procedure went wrong because of misconfiguration, you had better start all over again from the configuration step (see Section 4.4.1 [Configuration], page 56). The configuration script will ask you whether to clean up the system for reconfiguration. Please answer affirmative then.

Dependency rules written in Makefiles are inappropriate for using parallel make features provided by some versions of `make`. The compilation procedure of KLIC relies on the fact that contents of atom and functor databases are monotonically increasing. Dependencies on them are intentionally omitted to avoid redundant recompilation. Use the parallel compilation feature of the compiler driver `klic` that understands the mechanism instead. Parallelism used during installation procedure is specified at the configuration step (see Section 4.4.1 [Configuration], page 56).

If you think the problem is due to the distributed code, please report your problem to the following address.

`klic-bugs@icot.or.jp`

Including information on your host system (hardware and operating system), your configuration (contents of the file `config.sh`), and log of your installation would be of great help in analysing your problems.

### 4.5 Distributed Memory Parallel Implementation of KLIC

A version of the distributed parallel implementation of KLIC is included in this KLIC distribution. The distributed implementation is based on PVM 3.3. Implementations on other portable parallel processing libraries, such as MPI, and those on system-specific interprocess communication libraries have also been done, but not yet integrated into this distribution.

Although it is based on PVM, the current version does not support heterogeneous configuration: It does not work with systems consisting of processors with multiple architectures or running different operating systems. Currently, we don't have any plans to support heterogeneous systems.

#### 4.5.1 Installation of Distributed KLIC

To install the PVM version of the distributed KLIC, you have to first answer affirmatively to the question from the configuration script asking whether to configure for parallel implementations and then affirmatively again to the question asking whether to configure for

the distributed KLIC. Then it will ask for several questions on which directories the PVM system is installed and which PVM library is to be used, if several of them are available.

The following will be asked.

- Root directory of the pvm system
- The keyword for architecture of the system (**SUN4MP**, for example)
- The name of the PVM library (**pvm3**, for example)

The current version has problems with PVM implementations which does not use daemon processes. For example, on shared-memory multiprocessor Sparc systems running Solaris 2, the library **pvm3** does not work. Use **pvm3s** that use sockets instead of shared-memory for interprocess communication.

The rest of the installation procedure is the same as the procedure without the distributed KLIC system.

The distributed KLIC system runs exactly the same as its sequential version when the option for distributed processing (**-dp**) is not specified on compilation.

### 4.5.2 Compiling Programs for Distributed KLIC

Compilation procedure is almost the same for the sequential version except that the following option is available.

**-dp**            Specifies compilation for the distributed KLIC system. Without this option, the compiled object code will run only sequentially.

### 4.5.3 Running Programs of Distributed KLIC

#### 4.5.3.1 Setting Up PVM

Before executing programs compiled for distributed execution, the PVM system has to be running on your system. The following set up will be required.

- The following environment variables should be set properly.

**PVM\_ROOT**    The root directory of the PVM system installed on your system.

**PVM\_ARCH**    The keyword specifying the architecture of the system.

They should be the same as what you specified on installation of the KLIC system.

- The PVM demon should be running. The demon can be started by invoking the PVM console, which is in **\$PVM\_ROOT/lib/\$PVM\_ARCH/pvm**. It would be convenient to keep a window for this console.

For other setting parameters and details of operation of PVM console, please consult its own manual.

#### 4.5.3.2 Runtime Options for Distributed KLIC

The following options are available when running programs in the distributed KLIC system, in addition to those available for the sequential version.

**-p N**            Specifies the number of workers (Unix processes) for running the program.



- e** Specifies eager transfer mode. Normally, KLIC transfers data structures between processors on demands. Thus, nested data structures are transferred one level at a time. In the eager transfer mode, nested structures are sent at a time as far as they are already defined. This makes the execution more efficient for some programs, but may degrade the performance for others.
- E Level** Specifies how many level of nested data structures are to be transferred at each communication.
- I MicroSec** Specifies interval between interprocessor communication polling. Whether such polling is needed and which value to be appropriate depend on host systems and implementations of the physical communication layer. In most cases, its default value of 10000 is appropriate.
- n** Specifies printing out of some runtime statistics on interprocess communication.
- notimer** Specifies not to use timer-driven communication polling. Whether such polling is mandatory depends on implementation of the physical communication layer.
- relsp** Specifies that relative path should be used for the executable file on spawning worker tasks.
- S** Specifies not to notify receiver processes of communication packets by sending singals. On some implementations, this may speed up program execution by eliminating signal sending overheads.

#### 4.5.3.3 Known Bugs of Distributed KLIC

- Atoms and functors newly registered during program execution may not be handled properly.
- Specification of spying (see Section 4.3.5 [Spying], page 53) is effective only within the computation node where it is specified.

## 4.6 Shared-Memory Implementation of KLIC

A version of the shared-memory parallel implementation of KLIC is included in this KLIC distribution. The implementation contains hardware, operating system, and C compiler dependent parts. The version included is for Sparc-based systems running SunOS 5.3 and Alpha-based systems running DEC OSF/1. Gnu CC should be used for their compilation.

### 4.6.1 Installation of Shared-Memory KLIC

To install the shared-memory parallel version of KLIC, you have to first answer affirmatively to the question from the configuration script asking whether to configure shared-memory parallel implementation.

The rest of the installation procedure is the same as the procedure without the shared-memory KLIC system.

The shared-memory KLIC system runs exactly the same as its sequential version when the option for shared-memory parallel processing (**-shm**) is not specified on compilation.

### 4.6.2 Compiling Programs for Shared-Memory KLIC

Compilation procedure is almost the same for the sequential version except that the following option is available.

**-shm**            Specifies compilation for the shared-memory KLIC system. Without this option, the compiled object code will run only sequentially.

### 4.6.3 Running Programs of Shared-Memory KLIC

#### 4.6.3.1 Runtime Options for Shared-Memory KLIC

The following options are available when running programs in the shared-memory KLIC system, in addition to those available for the sequential version.

**-p *N***            Specifies the number of workers (Unix processes) for running the program.

**-D**                Reports process numbers of children workers. Maybe useful for lower level debugging.

**-S *Size***        Specifies the size of the shared heap. In the current implementation, shared heap is allocated at the initiation and will never be expanded.

#### 4.6.3.2 Known Bugs of Shared-Memory KLIC

- The tracer may not work correctly.

## Data Type Index

### A

atom..... 17

### F

floating point number..... 20

functor..... 23

### I

integer..... 19

### L

list..... 24

### M

merger..... 26

module..... 31

### P

predicate..... 31

### S

string..... 28

### V

vector..... 27

# Predicate, Method and Message Index

## \$

\$:= on builtin .....	21
\$< on builtin .....	23
\$:= on builtin .....	22
\$=< on builtin .....	23
\$=\ on builtin .....	23
\$> on builtin .....	22
\$>= on builtin .....	22

## :

:= on builtin .....	19
---------------------	----

## <

< on builtin .....	20
--------------------	----

## =

= on builtin .....	15
=.. on functor_table .....	24
:= on builtin .....	20
=< on builtin .....	20
=\ on builtin .....	20

## >

> on builtin .....	20
>= on builtin .....	20

## @

@< on builtin .....	16
@=< on builtin .....	16
@> on builtin .....	16
@>= on builtin .....	16

## \

\= on builtin .....	16
---------------------	----

## A

accept on bound socket .....	34
access on unix stream .....	35
acos on float .....	22
add on float .....	22
add on timer .....	43
addop on Prolog-like I/O .....	40
append_open on klicio stream .....	39
append_open on unix stream .....	33
apply on predicate .....	32
arg on builtin .....	24
argc on unix .....	36
argv on unix .....	36
arity on predicate .....	32
asin on float .....	22
atan on float .....	22
atom on builtin .....	18
atom_number on atom_table .....	18
atomic on builtin .....	17

## B

bind on unix stream .....	34
---------------------------	----

## C

call on predicate .....	32
cd on unix stream .....	34
ceil on float .....	22
chmod on unix stream .....	35
compare on builtin .....	15
compare on timer .....	43
connect on unix stream .....	34
cos on float .....	22
cosn on float .....	22
current_node on builtin .....	16
current_priority on builtin .....	16

## D

divide on float .....	22
-----------------------	----

## E

element on string .....	3, 30
element on vector .....	28
element_size on string .....	30
equal on float .....	23
exit on unix .....	36
exp on float .....	22

## F

fclose on C-like I/O.....	37
fclose on Prolog-like I/O.....	39
feof on C-like I/O.....	37
feof on Prolog-like I/O.....	39
fflush on C-like I/O.....	38
fflush on Prolog-like I/O.....	41
float on builtin.....	21
float on float.....	21
floor on float.....	22
fork on unix stream.....	36
fork_with_pipes on unix stream.....	36
fread on C-like I/O.....	38
fread on Prolog-like I/O.....	40
fseek on C-like I/O.....	37
fseek on Prolog-like I/O.....	39
ftell on C-like I/O.....	37
ftell on Prolog-like I/O.....	39
functor on builtin.....	24
fwrite on C-like I/O.....	38
fwrite on Prolog-like I/O.....	41

## G

gc on system_control.....	43
get_atom_name on atom_table.....	18
get_atom_string on atom_table.....	18
get_time_of_day on timer.....	43
getc on C-like I/O.....	38
getc on Prolog-like I/O.....	40
getenv on unix stream.....	36
gett on Prolog-like I/O.....	40
getwt on Prolog-like I/O.....	40
greater_than on float.....	23

## H

hash on builtin.....	16
----------------------	----

## I

instantiate_after on timer.....	43
instantiate_at on timer.....	43
instantiate_every on timer.....	44
integer on builtin.....	19
intern on atom_table.....	18

## J

join on string.....	31
join on vector.....	28

## K

kill on unix stream.....	36
klicio on klicio.....	39

## L

less_than on float.....	23
less_than on string.....	30
linecount on C-like I/O.....	38
linecount on Prolog-like I/O.....	40
list on builtin.....	24
log on float.....	22

## M

make_atom on atom_table.....	18
mktemp on unix stream.....	35
module on module.....	31
module on predicate.....	32
multiply on float.....	22

## N

name on module.....	31
name on predicate.....	32
new on float.....	21
new on merge.....	26
new on module.....	31
new on predicate.....	32
new on random_numbers.....	44
new on string.....	29
new on vector.....	27
new_functor on builtin.....	24
new_string on builtin.....	29
new_vector on builtin.....	27
nl on Prolog-like I/O.....	41
not_equal on float.....	23
not_greater_than on float.....	23
not_less_than on float.....	23
not_less_than on string.....	30
Number on C-like I/O.....	38
Number on Prolog-like I/O.....	41

## P

postmortem on system_control.....	42
pow on float.....	22
predicate on predicate.....	32
putc on C-like I/O.....	4, 38
putc on Prolog-like I/O.....	41
putenv on unix stream.....	36
putt on Prolog-like I/O.....	40
puttq on Prolog-like I/O.....	40
putwt on Prolog-like I/O.....	40
putwtq on Prolog-like I/O.....	40

## R

read_open on klicio stream.....	39
read_open on unix stream.....	33
rmop on Prolog-like I/O.....	40

## S

search_character on builtin.....	31
search_character on string.....	31
set_element on string.....	30
set_element on vector.....	28
set_string_element on builtin.....	30
set_vector_element on builtin.....	28
setarg on builtin.....	24
signal_stream on unix stream.....	35
sin on float.....	22
sinh on float.....	22
size on string.....	30
size on vector.....	27
split on string.....	30
split on vector.....	28
sqrt on float.....	22
stderr on klicio stream.....	39
stderr on unix stream.....	33
stdin on klicio stream.....	39
stdin on unix stream.....	33
stdout on klicio stream.....	39
stdout on unix stream.....	33
string on builtin.....	30
string on string.....	30
string_element on builtin.....	3, 30
string_less_than on builtin.....	30
string_not_less_than on builtin.....	30
sub on timer.....	43
subtract on float.....	22
sync on C-like I/O.....	38
sync on Prolog-like I/O.....	39
system on unix stream.....	36

## T

tan on float.....	22
tanh on float.....	22
times on unix.....	37

## U

umask on unix stream.....	35
unbound on builtin.....	16
ungetc on C-like I/O.....	38
ungetc on Prolog-like I/O.....	40
unix on unix.....	32
unlink on unix stream.....	35
unwrap on variable.....	42
update_open on klicio stream.....	39
update_open on unix stream.....	33

## V

vector on builtin.....	27
vector on vector.....	27
vector_element on builtin.....	28

## W

wait on builtin.....	15
wrap on variable.....	42
write_open on klicio stream.....	39
write_open on unix stream.....	33

## Module Index

### A

atom\_table ..... 18

### F

functor\_table ..... 23

### G

generic (pseudo module) ..... 7

### K

klicio ..... 39

### S

system\_control ..... 42

### U

unix ..... 32

# Concept Index

## A

aborting .....	36
accumulator .....	11
alternatively .....	9
argument mode .....	4
argument pair .....	9
argument pair name .....	10
arithmetics on floating points .....	21
arithmetics on integers .....	19
array .....	27
asynchronous I/O .....	34
atom .....	17
atomic data .....	17

## B

body .....	5
body method .....	8
break point .....	53
bug report .....	4

## C

car .....	24
cdr .....	24
ceiling .....	22
character code .....	19
character input .....	38
character output .....	38
character string .....	28
chdir .....	34
class .....	3
clause .....	5, 6
clause preference .....	9
closing a file .....	37
command line arguments .....	36
comparison .....	15
comparison of strings .....	30
comparison on floating points .....	22
comparison on integers .....	20
compilation .....	45
concurrent logic programming language .....	5
configuration .....	56, 57, 59
cons cell .....	24
converting integer to floating point .....	21
copyright .....	1
cosine .....	21
creating generic objects .....	8
creation of floating point numbers .....	21
creation of vectors .....	27
C .....	12, 37

## D

debugging .....	48
decrement .....	11
depth limit of trace display .....	54
dictionary order .....	30
difference list .....	11
directory .....	34
distributed KLIC .....	57
distribution .....	1
dump .....	55

## E

end of file .....	37
environment variable .....	36
execution .....	5
exit code .....	36
expanded pair .....	10
exponential .....	21

## F

file .....	34
floating point arithmetics .....	21
floating point comparison .....	22
floating point conversion from an integer .....	21
floating point notation .....	21
floating point number .....	20
flooring .....	22
flushing changes .....	37
forking processes .....	36
functor .....	23
functor notation .....	23

## G

garbage collection .....	42
generic method .....	3, 7
generic object .....	7
GHC .....	5
goal .....	7
goal pool .....	55
guard .....	5
guard method .....	8

## H

hashing .....	15
header file .....	12
higher order .....	31
hyperbolic function .....	21



**I**

I/O .....	37
ICOT Free Software .....	1
increment .....	11
initial goal .....	7
inline .....	12
input .....	37, 38
input argument .....	4
installation .....	56
integer .....	19
integer arithmetics .....	19
integer comparison .....	20
integer to floating point conversion .....	21
interprocess communication .....	4
interrupt .....	35
interval timer .....	43

**K**

KL1 .....	5
-----------	---

**L**

length limit of trace display .....	54
linkage .....	45
Linux .....	34
list .....	24
logarithm .....	21

**M**

mailing list .....	4
main .....	7
merging .....	25
message .....	4
message sending .....	11
message stream .....	25
method .....	3, 7
module .....	6

**N**

negation .....	6
new release .....	4
nil .....	24
notation of lists .....	25

**O**

object creation .....	8
open .....	33, 39
operating system .....	32
operations on functors .....	23
operator precedence grammar .....	39
otherwise .....	6
output .....	37
output argument .....	4

**P**

paired argument .....	9, 10
parallel processing .....	57, 59
port .....	49, 53
postmortem processing .....	42
predicate .....	3, 6
preference of clauses .....	9
principal functor .....	16, 23
priority .....	8, 16
process forking .....	36
program .....	31
PVM .....	56, 57

**R**

random number .....	44
reading in .....	38
ready queue .....	55
real number .....	20
record structure .....	23
rounding .....	22
running .....	47

**S**

seek .....	37
shared-memory KLIC .....	59
shell command .....	36
signal .....	35
signal sending .....	36
sine .....	21
spy .....	53
square root .....	22
standard input .....	33, 39
standard order .....	15
standard output .....	33, 39
stdio .....	33
stream .....	25
string .....	28
string input .....	38
string output .....	38
structure .....	23
subterm .....	54
suspended goal .....	55
symbol .....	17
symbolic atom .....	17
synchronization .....	37

**T**

tangent .....	21
time .....	43
timer .....	43
trace display .....	54
tracing .....	48
trigonometric function .....	21

**U**

unbound .....	16
unification .....	15
Unix interface .....	32
unlink .....	35
update .....	11

**V**

vector .....	27
verbose print .....	54

**W**

wrapped term .....	41
writing out .....	38