TM-0949

KL1 Programming

by
K. Susaki

August, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

KL1 Programming

Kasumi Susaki

Contents

Chapter 1

Introduction

This manual is intended to give novice users an introduction to develop programs in KL1 assuming that they know how to write simple programs in Prolog.

For more information, please refer to the following manuals.

- PIMOS 1.5 Introductory Manual
  - How to use the basic functions of PIMOS.
  - How to use I/O devices from user programs.
- PIMOS 1.5 Operating Manual
  Describes all the functions of PIMOS version 1.5.
- KL1 Programming Manual: This manual includes an Introduction, Beginning Level, and Intermediate Level.

Chapter 2

The KL1 Language

KL1 is a parallel logic programming language that has been developed at ICOT. It is based on the Flat version of GHC (Guarded Horn Clause) but with some extensions to make it a more practical language. The operating system for the parallel inference machine is written entirely in KL1.

2.1 Syntax and basic concepts

KL1 introduces the concept of **modules** and it is necessary to declare the name of the module at the top of the program. Public declaration is also necessary to specify the predicates that can be referred from outside the module. Here is a small example that returns the negation of input data in the boolean algebra.

```
%% sample 1

:- module not.
:- public not/2.
        not(In, Out):- In =:= 0 | Out = 1.      % (1)
        not(In, Out):- In =:= 1 | Out = 0.      % (2)
```

The first line of this program is the module declaration. ":- module" is the reserved word to show the module declaration and the following "not" is the specified name of this module. The second line is the specification of a public predicate. In this program, 'not/2' is specified as a public predicate. The following two lines are the program body.

In general, KL1 programs take the following form :

$$\underbrace{\underbrace{\underbrace{not(In,Out)}_{head} :- \underbrace{In\ -:= 0}_{goal}}_{guard} \underbrace{\ |\ }_{commit} \underbrace{\underbrace{Out = 1.}_{goal}}_{body}}_{clause}$$

Each KL1 clause is divided into two parts by an operator, | , called the "commit operator". To the left of this commit operator is the "guard" part, which contains the head of clause and some goals. To the right of the commit operator is the "body" part. In the guard part, only some specially prepared built-in predicates can be used. In the body part, we may use both built-in predicates for the body part and user-defined predicates.

In the guard part, the goals are executed in the order of appearance. In the body part, the goals may be executed concurrently.

All clauses with the same head are in an OR-relationship, as in Prolog. The predicate succeeds if the first clause is true, OR the second clause is true, OR ... and so on. In Prolog the order of clauses has meaning. That is, each clause is tried in the order in which it appears in the program. After selecting one clause, the execution can backtrack.

In KL1, all clauses are tried concurrently and the order does not have meaning in execution time. The goal that all guard goals are succeeded is selected (we call it commitment) and after that, the selection cannot backtrack. In this way, the commitment operator is similar to the cut operator in Prolog. If there are some alternatives that can be selected, the KL1 system selects and executes one clause at random from those whose guard part succeeds, and discards the rest.

If, in the previous example, illegal data is input (for example In = 2), this program fails (reduction failure). When we want to handle illegal data, one more clause is needed.

```
%% sample 2

:- module not.
:- public not/2.
```

```
not(In, Out):- In =:= 0 | Out = 1.                   % (1)
not(In, Out):- In =:= 1 | Out = 0.                   % (2)
not(In, Out):- In =\= 0, In =\= 1 | Out = -1. % (3)
```

(3) is the clause for illegal data.

```
not(In, Out):- true | Out = -1.          (3')
```

If the clauses are tried one by one in the order in which they appear (like in Prolog), we can write the third clause like (3'). But in KL1, (3') can be tried before (1) and (2), and the system can select this clause when In = 1 or In = 0.

2.2 negation

There is a special statement **otherwise**, a simple expression indicating negation. If otherwise is inserted between clauses, the KL1 system selects the clauses after the **otherwise** statement only when all clauses before it fail. Using the otherwise statement means that it is not necessary to describe the negative conditions of the guard parts of all the other clauses. The previous sample program can be rewritten with **otherwise** as follows :

```
%% sample 3

:- module not.
:- public not/2.
        not(In, Out):- In =:= 0 | Out = 1.      % (1)
        not(In, Out):- In =:= 1 | Out = 0.      % (2)
otherwise.
        not(In, Out):- true | Out = -1.         % (3)
```

Chapter 3

Unification and Synchronization

In the previous section, we saw how each clause of KL1 is divided into two parts: guard and body. After the success of guard goals, the clause is selected and the body goals of that clause may be executed. In other words, in the guard part, the conditions are checked and the system decides whether to select that clause or not.

In KL1, the unification in the guard and in the body are treated as different operations: guard unification and body unification.

3.1 guard unification

To check a condition, it is necessary to have a concrete value. So if the variable does not instantiated yet, the execution stops and waits for instantiation. We call this situation **suspension**. It means that in the guard part, no variable may be instantiated and concrete values are compared. This is called 'guard unification' or 'passive unification'. This mechanism, very different from Prolog, is explained with the following example.

```
%% example 1

?- not:not(Input,Result).

not(In,Out):- In =:= 0 | Out = 1.
not(In,Out):- In =:= 1 | Out = 0.
```

This goal waits the instantiation of 'Input' and after instantiation the guard goal will be instantiated. But in this example, there is no operation that gives a value to the variable 'Input', and this goal falls into perpetual suspention (deadlock).

```
%% example 2

?- not:not(Input,Result), Input = 1.

not(In,Out):- In =:= 0 | Out = 1.
not(In,Out):- In =:= 1 | Out = 0.
```

In this case, when Input is unified with '1' as the second goal, the goal 'not' resumes.

```
%% example 3

?- foo:through(abc,abc).

through(In,Out):- In = Out | true.
```

Because both arguments are concrete values, the guard goal 'In = Out' can be achieved. This example is the same as example 4.

```
%% example 4

?- foo:through(abc,abc).

through(Same,Same):- true | true.
```

These two variables have the same name 'Same'. This expression is the same as the guard goal 'A = B'.

In Prolog execution, the following (fifth) Example would be the same as the previous two examples, but in KL1 it behaves differently.

```
%% example 5

?- foo:through(Abc, Abc).

through(Same,Same):- true | true.
```

This invocation has the same variable name for two arguments. But the goal 'through/2' needs the same value (instantiated value) to execute. Before instantiation of the variable 'Abc', the goal should be suspended.

In the following example, the execution should also be suspended.

```
%% example 6

?- foo:through(In, Out).

through(Same,Same):- true | true.
```

In KL1, guard unification between variables should be suspended.

3.2 Body unification

In the body part, a variable may be instantiated like in Prolog and two different variables may be unified. This unification is called 'body unification' or 'active unification'.

In the previous example 'not', the variable 'Out' in the body part is instantiated to 1 or 0 or -1.

3.3 Synchronization with Unification

Here is a small example. This program generates integers from 0 to 100 and sieves them into odd and even numbers. It is a typical example of synchronization in KL1.

```
%% sieve

:- module sieve.
:- public go/2.
        go(Even,Odd):- true |                                   % (1)
                generate(0,Numbers),                            % (2)
                sieve(Numbers,Even,Odd).                        % (3)
        generate(N,Numbers):- N > 100 | Numbers = [].           % (4)
        generate(N,Numbers):- N =< 100 |                        % (5)
                Numbers = [N | NewNumbers],                     % (6)
                N1 := N + 1,                                    % (7)
                generate(N1,NewNumbers).                        % (8)
        sieve([H|T],Even,Odd):- (H mod 2) =:= 0 |               % (9)
                Even = [H|NewEven],                             % (10)
                sieve(T,NewEven,Odd).                           % (11)
        sieve([H|T],Even,Odd):- (H mod 2) =:= 1 |               % (12)
                Odd = [H|NewOdd],                               % (13)
                sieve(T,Even,NewOdd).                           % (14)
        sieve([],Even,Odd):- true |                             % (15)
                Even = [],                                      % (16)
                Odd = [].                                       % (17)
```

The goal number (1) is the top goal. When we invoke this goal, the even and odd numbers from 0 to 100 are returned in the first and second arguments.

The top goal 'go/2' makes two subgoals 'generate/2' and 'sieve/3'. 'generate/2' generates the integers from 0 to 100 by using tail recursive call. This goal unifies the generated number to the argument 'Numbers' with the form of list.

Line (9) to (17) are the definition of the clause 'sieve/3'. the clauses on lines (9) and (12) wait for the instantiation of the first argument. When the first argument is instantiated to a list, the first element of list (car) is checked. That is, generate/2 makes one integer number and sends it to sieve/3. Then it runs one procedure for that number and waits for the arrival of the next number.

Chapter 4

Operations on Built-in Data Types

4.1 Built-in data types

KL1 prepares the following data types.

- atom, integer, list, variable $\longrightarrow$ the same as Prolog
- vector $\longrightarrow$ random access structure
- string $\longrightarrow$ random access structure whose elements should be character codes
- module, code $\longrightarrow$
  - module :: a block of object code for KL1 programs
  - code :: entries of individual predicates

  They are treated in the same way as other data objects.

4.2 Strings

String are random access structures whose elements should be character codes. The elements can be read and updated by designating their position.

There are four kinds of string (their element sizes are different):

1-bit, 8-bit, 16-bit, 32-bit.

The default element size is 16-bit (**ascii** code is treated as 8-bit string). The string which has default element size can be written as follows :
"abc", "", string#"abc", and so on.

built-in predicates

KL1 system provides the following built-in predicates for strings. The predicate with ':: G' is for the guard part, and that with ':: B' is for the body part.

1. Check the data type

```
string(String, ^Size, ^ElementSize) :: G
string(String, ^Size, ^ElementSize, ^NewString) :: B
```

2. Create new string

```
new_string(^String, Size, ElementSize) :: G
```

3. Read the element

```
string_element(String, Position, ^Element) :: G
string_element(String, Position, ^Element, ^NewString) :: B
```

4. Update the element

```
set_string_element(String, Position, NewElement, ^NewString) :: B
```

Example

Here is a small example. This program appends two strings.

```
%% append strings

:- module string.
:- public append/3.

append(Str1,Str2,Str):- string(Str1,S1,SE), string(Str2,S2,SE) |
        S12 := S1 + S2,
        new_string(NStr, S12, SE),
        appendArgs(0,S1,0,Str1,NStr,Nstr1),
        appendArgs(0,S2,S1,Str2,NStr1,Str).

appendArgs(N,N, _, _, Str,NStr):- true | NStr = Str.
appendArgs(M,N,SM, OStr, Str, Nstr):- M < N |
        string_element(OStr,M,E) |
        set_string_element(Str,SM,E,Str1),
        N1 := M + 1, SM1 := SM + 1,
        appendArgs(M1,N,SM1,OStr,Str1,NStr).
```

4.3 Vectors

Vectors are random access structures whose elements may be arbitrary data types. A variable also may be an element of a vector. Vectors are written with curly brackets. There is also a functor format vector as in DEC-10 Prolog.

{a,b,c}, {}, a(X,Y), and so on.
{a,b,c} is the same as a(b,c).

Built-in predicates

The KL1 system provides the following built-in predicates for vector.

1. Check the data type

```
vector(Vector, ^Size) :: G
vector(Vector, ^Size, ^NewVector) :: B
```

2. Create new vector

```
new_vector(^Vector, Size) :: B
```

3. Read vector element

```
vector_element(Vector, Position, ^Element) :: G
vector_element(Vector, Position, ^Element, ^NewVector) :: B
```

4. Update vector element

```
set_vector_element(Vector, Position, ^OldElem, NewElem, ^NewVect) :: B
```

4.4 Streams

In the sample program 'sieve', we use a list for an arbitrary number of communication between two processes. This popular programming technique is called 'stream programming'.

When many processes correspond with one stream (many processes wait for a message on the same stream and there is one process that is sending messages on that stream), there should be a process that merges those streams.

We call this process 'merger' and, for efficiency, it is prepared as a built-in predicate.

```
merge(In,Out)  :: B
```

The input stream 'In' can be instantiated with vector; this merger is so flexible that it can have an arbitary number of input streams.

See the following example.

```
%% sample 4

:- module counter.
:- public counter/1.
counter(In):- true |
        merge(In,Out),
        counter(Out,0).
counter([up|In],State):- true |
        New := State + 1,
        counter(In,New).
counter([down|In],State):- true |
        New := State - 1,
        counter(In,New).
counter([show(Current)|In],State):- true |
        Current = State,
        counter(In,State).
counter([],State):- true | true.
```

In this program, merge(In,Out) is invoked in the initiation of this program. And the program that invokes this module may split the input stream to 'counter' and use those streams independently as follows.

```
%% sample 5

:- module calculator.
:- public go.
go:- true |
        calculatorA(A),
        calculatorB(B),
        In = {A,B},
        counter:counter(In).
```

In this example, both calculatorA and calculatorB have the input stream to 'counter'. And they can send messages independently. When they send messages at the same time, which message (from A or B) arrives first is not determined.

Chapter 5

Process Oriented Programming

Process oriented (object oriented) programming in KL1 is a programming technique like the previous example 'sieve/2'. Process is realized by a tail-recursive call goal, there are some connections between processes by streams, and they communicate with each other by sending messages. A program is described as a process network.

In the previous section, counter/1 is a process. It receives the messages 'up', 'down' and 'show(Current)' and in the cases of 'up' or 'down' it changes the state. 'show(Current)' is a different kind of message. When the counter receives this message, it returns its state in the argument 'Current'. The argument of message 'show' is used to return the value to the sender of this message. In this way, it is not necessary to have another line for answer messages, and it is not necessary to know the sender of the message either. We call this technique 'back communication'.

Chapter 6

Execution Control

The execution order of goals is decided only by the dependency of the data. It is not enough for sophisticated problem solving, and it is necessary to have functions to control the execution precisely. KL1 provides two functions to control the execution, they are:

- Priority Control
- Load Distribution

6.1 Priority control

For specifying a sophisticated problem-solving strategy that can use the available computational resources effectively, it is essential to introduce the notion of priority for goals that can be executed in parallel and for clauses that can be chosen non-deterministically. To write the operating system, the priority for the sets of goals is also introduced with the function of called **shoen**, which is described in the next section.

6.1.1 Priority of goals

An order of priority is assigned to each goal. Without priority specification, a goal has the same priority as its parent goal. The priority is specified in relative way. There are two ways of doing this relative specification.

- Specifying the priority of a goal by the ratio between minimum and maximum priority of that goals's shoen (ratio specification within the assigned shoen).
- Specifying the priority of a goal (child) by the ratio between the priority of a goal (parent) that is calling the child and the minimum (or the maximum) priority of the shoen that surrounds the child goal (self-relative specification within the child goal's shoen).

Ratio specification within the assigned shoen

Each goal has its own maximum and minimum priority associated with the shoen in which the goal executes. In this way, the priority for the goal is specified by the ratio between these two values.

The specification is written as follows :

```
Goal@priority(*,Ratio)
```

The priority for this goal is decided with following formula.

$$Cp + (\text{Maximum} - Cp) \times \frac{|\text{Ratio}|}{4096} \qquad (0 \leq \text{Ratio} \leq 4096)$$

or

$$Cp - (Cp - \text{Minimum}) \times \frac{|\text{Ratio}|}{4096} \qquad (-4096 \leq \text{Ratio} < 0)$$

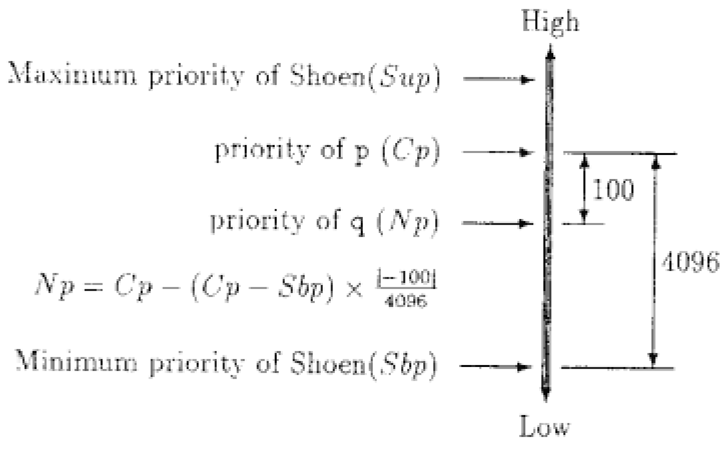
Self-relative specification within the goal's shoen

Each goal has the priority of its parent goal as a default value. In this way, the priority is specified by the ratio between its default and the minimum priority of the shoen in which the goal executes.

```
Goal@priority($,Ratio)
```

The priority for this goal is decided with the following formula.

$$\text{Minimum} + (\text{Maximum} - \text{Minimum}) \times \frac{\text{Ratio}}{4096} \qquad (0 \leq \text{Ratio} \leq 4096)$$


High
Maximum priority of Shoen(Sup)
priority of p (Cp)
100
priority of q (Np)
4096
$Np = Cp - (Cp - Sbp) \times \frac{|-100|}{4096}$
Minimum priority of Shoen(Sbp)
Low

6.1.2 Priority of clauses

The 'alternatively' statement shows the priority of clause. If inserted between clauses, the clauses before the statement have higher priority than those that follow it. That is, clauses that appear after the 'alternatively' statement will be selected if all the previous clauses fail or suspend.

Consider the following program.

```
        m([W | X], Y, WZ) :- true |
                WZ = [W | Z], m( X, Y, Z).       (1)
        m(X, [W | Y], WZ) :- true |
                WZ = [W | Z], m(X, Y, Z).        (2)
```

In this program, two clauses wait different arguments. When we want to specify higher priority for the first argument, use the statement **alternatively.**

```
        m([W | X], Y, WZ) :- true |
                WZ = [W | Z], m( X, Y, Z).       (1)
alternatively.
        m(X, [W | Y], WZ) :- true |
                WZ = [W | Z], m(X, Y, Z).        (2)
```

But the specification with alternatively is not an absolute one. The program should run correctly without this statement. It should be used only for efficiency.

6.2 Load Distribution

In the current version of PIMOS, a processor number must be explicitly assigned to a goal in order to make it execute on a different processor. Without the specification, the goals are executed in the same processor that invokes them.

Any KL1 body goal except for built-in predicate may be assigned to a specific processor by attaching an expression containing the PE number in the following manner, where PE is an integer equal to or greater than zero.

```
        goal@processor(PE)
```

The following sample program illustrates this concept.

```
%% sample 6

        :- module distribution.
        :- public foo/0.

        foo:- true |
                a@processor(0),
                a@processor(1),
                a@processor(2).
        a :- true | true.
```

Processor numbers may also be variables. Execution of the corresponding goal is suspended until the variable is instantiated to a number.

```
%% sample 7

        :- module distribution2.
        :- public foo/3.

        foo(P1,P2, P3):- true |
                a@processor(P1),
                a@processor(P2),
                a@processor(P3).
        a :- true | true.
```

There is a built-in predicate 'current_processor(PE,X,Y)' that returns the number of the processor on which it was executed along with the number of processors available in the horizontal(X) and vertical(Y) directions.

Chapter 7

Shoen

Shoen helps to write an operating system and is a function to control the execution of a set of goals (a goal and its subgoals), for example to control a job that runs on shell. It is not intended for use by application programmers, but it is also helpful for them. So in this chapter, I will give a brief introduction to shoen.

7.1 The Shoen feature

Shoen is a unit provided as a KL1 language primitive to handle the following things.

- Execution control
- Resource management
- Exceptional events

Shoen has two streams. To control the execution of shoen, send messages via the **control stream**. To report the internal status of shoen, shoen uses the **report stream**.

Shoen is created with a following built-in predicate and shoen can be nested.

```
shoen:execute(Code,Argv,MinPrio,MaxPrio,Mask,Control,^Report)
```

Code, Argv

The top goal that is executed inside the creating shoen is designated by **Code** and **Argv**.

MinPrio, MaxPrio

The priority for this shoen is assigned with **MinPrio** and **MaxPrio**. These arguments may have integer values from 0 to 4096. These values are not practical ones. They show the ratio between the goal that invokes this built-in predicate and the minimum value outside the shoen.

Mask

This is a mask pattern. This pattern decides the exceptional events that this shoen monitors.

Control, Report

Control is the control stream of this shoen and **Report** is the report stream of this shoen.

7.2 Execution control

By sending messages to the shoen by the control stream, we can start, stop and abort the execution of goals inside the shoen. Corresponding to these messages, acknowledgements are returned by report stream.

7.3 Resource Management

Shoen is a unit for resource management. Usually an operating system manages resource (for example CPU time, memory size), but in PIMOS, it is difficult to manage CPU time and memory. So in the current version, roughly speaking, PIMOS manages the number of reductions as a resource.

To assign the maximum number of resources, we also send a message by control stream. Before the number of consumed resources reaches the maximum, a message reports the shortage by the report stream. You can add an amount of resources also by sending a message.

7.4 Exception handling

In KL1, all the goals are in And-relation. That is, when a user program fails, the failure propagates to all the other goals and PIMOS itself fails. When a goal inside a shoen fails, the failure propagates only inside that shoen. It does not have any effect outside the shoen.

The shoen monitors exceptional events such as failure (reduction failure or unification failure), deadlock and so on. These exceptional events are reported as messages by the report stream.

7.5 Cautions

It is very difficult to use the shoen function when there is a variable shared between the inside of the shoen and the outside. Another drawback is that the overhead to create a shoen is very big.

Chapter 8

MRB scheme

8.1 MRB scheme

The **MRB** (Multiple Reference Bit) scheme is a reference counting scheme for incremental garbage collection. Roughly speaking, the MRB scheme is as follows:

In the MRB scheme the bit is used to discriminate the single referenced data from others. The single referenced data is managed in special manner for efficiency (for real time garbage collection).

It is better to keep the MRB white for efficient use of memory area and for efficiency of execution.

In other words, MRB shows whether the data is referred from only one goal or from more that one goal. The pointer to that data is white when there is only one goal and black when there are more that one goal.

The rule to determine whether a path is white or black is as follows :

case 1 : The pointer to a variable is white when the reference of that variable must be less than two. Otherwise it is black.

case 2 : The pointer to a concrete value is white when the reference of that data must be one. Otherwise it is black.

The value of the MRB is determined when a goal commits. Please see the following example.

```
?- p(X), q(X).
```

```
p(X) :- true | r(X), s(X).      (1)
r(X) :- true | true.            (2)
```

When these goals are invoked, the variable X is uninstantiated and the occurrence is twice, so the pointer is white. After the execution of (1), the number of reference becomes three and the MRB changes to black. After executing (2), the reference decreases to two. But the MRB is black and does not change.

8.2 set_vector_element/5

In the previous section, you see the built-in predicate set_vector_element/5.

```
set_vector_element(Vect,Position,Oldelement,NewElement,NewVector)
```

This works as follows :

Replace the element of 'Vect' whose position is 'Position' with 'NewElement'. To do this, this goal makes new vector 'NewVector' with the 'NewElement'.

It is for keeping the single reference for the target vector. The update operation for vector is done in an efficient way when the reference is single.

Chapter 9

Macro

Several categories of macro are introduced in KL1. They are as follows:

- Macros for the description of constants.
  string#"abc", key#lf and so on.
- Macros for arithmetic comparison.

```
        X > Y, X =\= Y and so on.
```

- Macros for conditional branch.
- Macros for the declaration of implicit arguments.

To use these macros, it is necessary to put in the following statement in the top of the module definition.

User defined macros have not been introduced yet.

Conditional branch macros

In KL1 the macros to write conditional branch are provided. Here is a small example and the generated code.

```
foo(X,Y) :- true |
      ( X=:=0 -> p(Y,Z);
        X > 0 -> q(Y,Z);
        otherwise.
        true -> r(Y,Z) ),
      s(X,Z).
```

The generated program is as follows.

```
foo(X,Y) :- true |
      '$foo/2/0'(X,Y,Z),
      s(X,Z).

'$foo/2/0'(X,Y,Z) :- X=:=0 | p(Y,Z).
'$foo/2/0'(X,Y,Z) :- X > 0 | q(Y,Z).
otherwise.
'$foo/2/0'(X,Y,Z) :- true | r(Y,Z).
```

A conditional branch macro may allow an arbitrary number of expressions in the following format with a semi-colon as a descriptor.

```
guard -> body
```

All built-in predicates or expressions that may be written in the guard part of a clause are available as the guard of this expression. Also, all the expressions in the body part of the clause are available in the body of this expression.

Chapter 10

Puzzles

10.1 How to use I/O operation

Any KL1 program can access a window of the Shell or the Listener by using the Standard I/O device represented by the streams Standard-Input, Standard-Output, Standard-Input/Output, Message-Output, and Standard-Interaction.

In order to perform I/O, messages are sent to these streams requesting operations such as "getc(C)" to read a character, "putc(C)" to output a character, and so on.

The following module shows how to access the streams Standard-Input, Standard-Output, and Message-Output.

```
%% sample 8

        :- module std_io.
        :- public create/3.

        create(Input,Output,Message):- true|
            shoen:raise(pimos_tag#shell,get_std_in,Input),
            shoen:raise(pimos_tag#shell,get_std_out,Output),
            shoen:raise(pimos_tag#shell,get_std_mes,Message).
```

This module can be called as :

```
?- std_io:create(Input,Output,Message),
   Input=[ ... ], Output=[ ... ], Message=[ ... ].
```

Which results in the following streams :

- Input : A Standard-Input device stream
  Accepts any message provided by a `buffer:input_filter`.
- Output : A Standard-Output device stream
  Accepts any messages provided by a `buffer:output_filter`.
- Message : A Message-Output device stream
  Accepts any messages provided by a `buffer:output_filter`.

An explanation of the buffer and the filter utilities can also be found in Section 3.5 of the "PIMOS 1.5 Operating Manual".

The following program display the message 'Hello' to the shell window (or listener window when you invoke this program from listener).

```
:- module std_io.
:- public go/0.

go:- true|
        shoen:raise(pimos_tag#shell,get_std_io,IO),
        IO = [putt('Hello'), nl].
```

10.2 Fibonacci sequence

Write a program to generate a fibonacci sequence. Fibonacci sequence is defined by the following formula.

$$a_1 = 1$$
$$a_2 = 1$$
$$a_n = a_{n-2} + a_{n-1}$$

10.3 stream compression

Write a program to eliminate the duplicate elements from input stream and output the result.

10.4 Prime number generator

Write a program to generate a prime number.

10.5 Matrix transposition

Write a program which transposes a matrix. The matrix is written by vector. For example, a matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

is written {{1,2,3},{4,5,6}}.

Appendix A

Sample programs

A.1 Fibonacci sequence

```
:- module fib.
:- public fib/2.

fib(M,S) :- true | fib(M,1,0,S).
fib(M,N1,N2,S):- M =:= 0 | S = [].
fib(M,N1,N2,S):- M>0 |
        N3 := N1 + N2,
        S = [N3 | NS],
        M1 := M - 1,
        fib(M1,N2,N3,NS).
```

A.2 Stream compression

```
:- module stream.
:- public compact/2.

compact([],Ys) :- true | Ys = [].
compact([X|Xs], Ys) :- true |
        Ys = [X | Ys1],
        filter(X,Xs,Zs),
        compact(Zs,Ys1).
```

```
filter(_,[],Ys) :- true | Ys = [].
filter(K, [K|Xs],Ys) :- true | filter(K,Xs,Ys).
filter(K, [X|Xs],Ys) :- K\=X |
        Ys = [X|Ys1], filter(K,Xs,Ys1).
```

A.3 Prime number generator

```
:- module prime.
:- public go/1.

go(Max) :- true | primes(Max,Ps), output(Ps).

primes(Max,Ps) :- true | gen(2,Max,Ns), sift(Ns,Ps).

gen(N0,Max,Ns0) :- N0 =< Max |
    Ns0 = [N0|Ns1], N1 := N0 + 1, gen(N1,Max,Ns1).
gen(N0,Max,Ns0) :- N0 > Max | Ns0 = [].

sift([P|Xs1],Zs0) :- true |
    Zs0 = [P|Zs1],
    filter(P,Xs1,Ys),
    sift(Ys,Zs1).
sift([],Zs0) :- true | Zs0 = [].

filter(P,[X|Xs1],Ys0) :- (X mod P) =\= 0 |
    Ys0 = [X|Ys1], filter(P,Xs1,Ys1).
filter(P,[X|Xs1],Ys0) :- (X mod P) =:= 0 |
    filter(P,Xs1,Ys0).
filter(P,[],Ys0) :- true | Ys0 = [].

output(Ps) :- true |
    shoen:raise(pimos_tag#shell,get_std_out,Out),
    output(Ps,Out).

output([X|Xs1],Out) :- true |
    Out = [putt(X), nl | Otail],
    output(Xs1,Otail).
output([],Out) :- true |
    Out = [].
```

A.4 Matrix transposition

```
:- module matrix.
:- public transpose/2.

transpose(M,TM) :- vector(M,NumberOfRow),
        vector_element(M,0,M0), vector(M0,NumberOfCol) |
        newMatrix(NumberOfCol,NumberOfRow,TM0),
        transpose(M,NumberOfCol,NumberOfRow,TM0,TM).

newMatrix(RowN,ColN,M) :- true |
        new_vector(M0,RowN),
        newMatrixArgs(0,RowN,ColN,M0,M).

newMatrixArgs(To,To,_,M,MM) :- true | MM = M.
newMatrixArgs(From,To,Size,M,NM) :- From < To |
        new_vector(Row,Size),
        set_vector_element(M,From,_,Row,M1),
        From1 := From + 1,
        newMatrixArgs(From1,To,Size,M1,NM).

transpose(M,RowN,ColN,TM,NTM) :- true |
        transposeArgs(0,RowN,ColN,M,TM,NTM).

transposeArgs(To,To,_,_,TM,NTM) :- true | NTM = TM.
transposeArgs(From,To,ColN,M,TM,NTM) :- From < To |
        set_vector_element(TM,From,R,NR,TM1),
        transposeArgsRow(0,ColN,From,R,NR,M,M1),
        From1 := From + 1,
        transposeArgs(From1,To,ColN,M1,TM1,NTM).
```

```
transposeArgsRow(To,To,_,Row,NRow,M,NM) :- true |
        NRow = Row, NM = M.
transposeArgsRow(From,To,ColN,Row,NRow,M,NM) :- From < To |
        set_vector_element(M,From,MRow,NMRow,M1),
        set_vector_element(MRow,ColN,E,0,NMRow),
        set_vector_element(Row,From,_,E,Row1),
        From1 := From + 1,
        transposeArgsRow(From1,To,ColN,Row1,NRow,M1,NM).
```