A micro-manual

of

the programming language

L O G L A N - 82

Basic constructs and facilities

Author: Antoni Kreczmar

Institute of Informatics, Warsaw University
March 1990

edited by A.Salwicki LITA Pau November 1990

Table of contents

LOGLAN-82 is a universal programming language designed at the Institute of Informatics, University of Warsaw. Its syntax is patterned upon Pascal's. Its rich semantics includes the classical constructs and facilities offered by the Algol-family programming languages as well as more modern facilities, such as concurrency and exception handling.

The basic constructs and facilities of the LOGLAN-82 programming language include:

1) A convenient set of structured statements,

2) Modularity (with the possibility of module nesting and extending),

4) Classes (as a generalization of records) which enable to define complex structured types, data structures, packages, etc.,

5) Adjustable arrays whose bounds are determined at run-time in such a way that multidimensional arrays may be of various shapes, e.g. triangular, k-diagonal, streaked, etc.,

6) Coroutines and semi-coroutines,

7) Prefixing - the facility borrowed from Simula-67, substantially generalized in LOGLAN-82 - which enables to build up hierarchies of types and data structures, problem-oriented languages, etc.,

8) Formal types treated as a method of module parametrization,

9) Module protection and encapsulation techniques,

10) Programmed deallocator - a tool for efficient and secure garbage collection, which allows the user to implement the optimal strategy of storage management,

11) Exception handling which provides facilities for dealing with run-time errors and other exceptional situations raised by the user,

12) Concurrency easily adaptable to any operating system kernel and allowing parallel programming in a natural and efficient way.

The language covers system programming, data processing, and numerical computations. Its constructs represent the state-of-art and are efficiently implementable. Large systems consisting of many cooperating modules are easily decomposed and assembled, due to the class concept and prefixing(i.e. inheritance).

LOGLAN-82 constructs and facilities have appeared and evolved simultaneously with the experiments on the first pilot compiler (implemented on Mera-400 Polish minicomputer). The research on LOGLAN-82 implementation engendered with new algorithms for static semantics, context analysis, code generation, data structures for storage management etc.

The LOGLAN-82 compiler provides a keen analysis of syntactic and semantic errors at compilation as well as at run time. The object code is very efficient with respect to time and space. The completeness of error checking guarantees full security and ease of program debugging.

1. Compound statements

Compound statements in LOGLAN-82 are built up from simple statements (like assignment statement e.g. x:=y+0.5, call statement e.g. **call** P(7,x+5) etc.) by means of conditional, iteration and case statements.

The syntax of conditional statement is as follows:

```
if boolean expression
then
  sequence of statements
else
  sequence of statements
fi
```

where "else part" may be omitted:

```
if boolean expression
then
  sequence of statements
fi
```

The semantics of conditional statement is standard. The keyword fi
allows to nest conditional statements without appearence of "dangling else" ambiguity.

Example:

```
if delta>0

then
  x2:=sqrt(delta)/a/2;
  if b=0
  then

    x1:=x2
  else

    x1:=-b/a/2+x2; x2:=x1-2*x2
  fi

else

  if delta=0

  then

    x1:=-b/a/2; x2:=x1
  else

    write(" no real roots")
  fi

fi
```

The statements in a sequence of statements are separated with semicolons (semicolon may end a sequence , and then, the last statement in the sequence is the empty statement).

The short circuit control forms are realized in LOGLAN-82 by the conditional statements with orif (or andif) list. A conditional
statement with orif list has the form:

orif

```
if wb1 orif wb2 ... orif wbk
then
  sequence of statements
else
  sequence of statements
fi
```

and corresponds somehow to a conditional statement:

```
if wb1 or wb2 ... or wbk
then
  sequence of statements
else
  sequence of statements
fi
```

The above conditional statement (without orif list) selects for execution one of two sequences of statements, depending on the truth value of the boolean expression:

```
wb1 or wb2 or ... wbk
```

which is always evaluated till the end. For the execution of the conditional statement with orif list the specified conditons wb1,...,wbk are evaluated in succession, until the first one evaluates to true. Then the rest of the sequence wb1,...,wbk is abandoned and "then part" is executed. If none of the conditions wb1,...,wbk evaluates to true "else part" is executed (if any).

Conditional statements with orif list facilitate to program those con_ditions, which evaluation to the end may raise a run-time error.

Example:

The execution of the statement:

if i>n **or** A(i)=0 **then** i:=i-1 **else** A(i):=1 **fi**

where the value of i is greater than n, and A is an array with upper bound n, will raise the run-time error. Then the user can write:

if i>n **orif** A(i)=0 **then** i:=i-1 **else** A(i):=1 **fi**

what allows to avoid this run-time error and probably agrees with his intension.

Conditional statement with andif list has the form:

```
if wb1 andif wb2 ...  andif wbk
then
  sequence of statements
else
  sequence of statements
fi
```

For the execution of this kind of statements, the conditions wb1,...,wbk are evaluated in succession, until the first one evaluates to false; then "else part" (if any) is executed. Otherwise "then part" is executed.

Iteration statement in LOGLAN-82 has the form:

do sequence of statements **od**

An iteration statement specifies repeated execution of the sequence of statements and terminates with the execution of the simple statement exit

Example:

```
s:=1; t:=1; i:=1;
do

  i:=i+1; t:=t*x/i;
  if abs(t) < 1.0E-10 then exit fi;
  s:=s+t
od;
```

If two iteration statements are nested, then double exit in the inner one terminates both of them.

Example:

```
r,x:=0;
do

  s,t:=1; i:=1; x:=x+0.2;
  do

    i:=i+1; t:=t*x/i;
    if i > n then exit exit fi; (* termination of both loops *)

    if t < 1 then exit fi;      (* termination of the inner loop *)
    s:=s+t
  od

od
```

In the example above simultaneous assignment statements are illustrated (e.g. r,x:=0) and comments, which begin with a left parenthesis immediately followed by an asterisk and end with an asterisk immediately followed by a right parenthesis.

Triple exit terminates three nested iteration statements, four exit terminates four nested iteration statements etc.

The iteration statement with while condition:
while

```
while boolean expression
```

```
do
  sequence of statements
od
```

is equivalent to:

```
do
  if not boolean expression then  exit  fi;
  sequence of statements
od
```

The iteration statements with controlled variables (for statements) have the forms:

```
for j:=wa1 step wa2 to wa3
do
  sequence of statements
od
```

or

```
for j:=wa1 step wa2 downto wa3
do
  sequence of statements
od
```

The type of the controlled variable j must be discrete. The value of this variable in the case of the for statement with to is increased, and in the case of the for statement with

downto is decreased. The
discrete range begins with the value of wa1 and changes with the step equal to the value of wa2. The execution of the for statement with to terminates when the value of j for the first time becomes greater than the value of wa3 (with downto when the value of j for the first time
becomes less than the value of wa3). After the for statement
termination the value of its controlled variable is determined and equal to the first value exceeding the specified discrete range. The values of expressions wa1, wa2 and wa3 are evaluated once, upon entry to the iteration statement. Default value of wa2 is equal 1 (when the keyword step and expression wa2 are omitted).

For or while statements may be combined with exit statement.

Example:

```
for j:=1 to n
do

  if x=A(j) then exit fi;
od
```

The above iteration statement terminates either for the least j, 1<=j<=n, such that x=A(j) or for j=n+1 when x=/=A(j), j=1,...,n.

To enhance the user's comfort, the simple statement repeat is provided. It may appear in an iteration statement and causes the current iteration to be finished and the next one to be continued (something like jump to CONTINUE in Fortran's DO statements).

Example:

```
i:=0;  s:=0;
do

  i:=i+1;
  if A(i)<0 then repeat fi; (* jump to od,iterations are contd.*)
  if i > m then exit fi;    (* iteration statement is terminated*)
  s:=s+sqrt(A(i));
od;
```

Just as exit, repeat may appear in for statement or while statement. Then the next iteration begins with either the evaluation of a new value of the controlled variable (for statement) or with the evaluation of the condition (while statement).

Case statement in LOGLAN-82 has the form:

```
case WA
  when L1 : I1
  when L2 : I2
   ...
  when Lk : Ik
  otherwise  I
esac
```

where WA is an expression , L1,...,Lk are constants and I1,..., Ik,I are sequences of statements.

A case statement selects for execution a sequence of statements Ij, 1≤j≤k, where the value of WA equals Lj. The choice otherwise covers
all values (possibly none) not given in the previous choices. The execution of a case statement chooses one and only one alternative (since the choices are to be exhaustive and mutually exclusive).

2. Modularity

Modular structure of the language is gained due to the large set of means for module nesting and extending. Program modules (units) are blocks, procedures, functions, classes, coroutines and processes. Block is the simplest kind of unit. Its syntax is the following:

```
block
  lists of declarations
begin
  sequence of statements
```

```
     end
```

The sequence of statements commences with the keyword begin (it may be omitted when this sequence is empty). The lists of declarations define the syntactic entities (variables, constants, other units), whose scope is that block. The syntactic entities are identified in the sequence of statements by means of names (identifiers).

Example:

```
     block

       const n=250;

       var x,y:real, i,j,k: integer, b: boolean;

       const m=n+1;

     begin

       read(i,j);          (* read two integers *)
       x,y:=n/(i+j);         (* simultaneous assignment *)
       read(c) ;           (* read a character *)
       b:= c = 'a';        (* 'a'  a character *)
       for k:= 1 to m

       do
         write(x+y/k:10:4);  (* print the value of x+y/k in the
           field of  10 characters, 4 digits after the point *)
       od
     end
```

In the lists of declarations semicolons terminate the whole lists, not the lists elements. Any declaration list must begin with the pertinent keyword (var for variables, const for constants etc.). The value of an expression defining a constant must be determinable statically (at compilation time).

Program in LOGLAN-82 may be a block or alternatively may be of the following form:

```
program name;

  lists of declarations
begin

  sequence of statements
end
```

Then the whole program can be identified by that name (the source as well as the object code).

A block can appear in the sequence of statements (of any unit), thus it is a statement. (Main block is assumed to appear as a statement of the given job control language.)

For the execution of a block statement the object of block is created in a computer memory, and then, the sequence of statements is performed. The syntactic entities declared in the block are allocated in its object. After a block's termination its object is automatically deallocated (and the corresponding space may be immediately reused).

The modular structure of the language works "in full steam" when not only blocks, but the other kinds of units are also used. They will be described closer in the following points.

Unit nesting allows to build up hierarchies of units and supports security of programming. It follows from the general visibility rules; namely, a syntactic entity declared in an outer unit is visible in an inner one (unless hidden by an inner declaration). On the other hand, a syntactic entity declared in an inner unit is not visible from an outer one.

Example:

```
program test;

  var a,b,c:real, i,j,k:integer;

begin

  read(a,b,c,i);
  block

    var j,k:real;
```

```
      begin
        j:=a; k:=j+b; write(" this is the inner block ",j,k)
      end;
      write(" this is the outer block ",i,a:20)
    end;
```

In this program, first the main block statement is executed (with variables a,b,c,i,j,k). Next, after the read statement, the inner block statement is executed (with variables j,k). In the inner block the global variables j,k are hidden by the local ones.

3. Procedures and functions

Procedures and functions are well-known kinds of units. Their syntax is modelled on Pascal's, though with some slight modifications. Procedure (function) declaration consists of a specification part and a body.

Example:

```
unit Euclid: function(i,j:integer):integer;
var k:integer;
begin
  do
    if j=0 then exit fi;
    k:=i mod j; i:=j; j:=k
  od;
  result:=i
end;
```

Procedure or function specification begins with its identifier preceded by the keyword unit. (The same syntax concerns any other
module named unit.) Then follows its kind declaration, its formal parameters (if any), and the type of the returned value (only for functions). A body consists of declaration lists for local entities and a sequence of statements. The keyword **begin** commences the sequence of statements, and is omitted, if this sequence is empty. The value returned by a function equals to the most recent value of the standard variable "result", implicitly declared in any function. This variable can be used as a local auxiliary variable as well.

Example:

```
unit Newton: function(n,m:integer):integer;

     var i:integer;
begin

  if m > n then return fi;

  result:=n;
  for i:=2 to m do result:=result*(n-i+1) div i od

end Newton;
```

The optional identifier at the end of a unit must repeat the identifier of a unit. It is suggested that the compilers check the order of unit nesting, so these optional occurrences of identifiers would facilitate program debugging.

All the local variables of a unit are initialized (real with 0.0, integer with 0, boolean with false etc.). Thus , for instance, the value of function Newton is 0 for m>n, since "result" is also initialized, as any other local variable.

The return statement (return) completes the execution of a procedure (function) body,i.e. return is made to the caller. If return does not
appear explicitly, return is made with the execution of the final end
of a unit. Upon return to the caller the procedure (function) object is deallocated.

Functions are invoked in expressions with the corresponding list of actual parameters. Procedures are invoked by call statement (also with the corresponding list of actual parameters).

Example:

```
i:=i*Euclid(k,105)-Newton(n,m+1);
call P(x,y+3);
```

Formal parameters are of four categories: variable parameters, procedure parameters, function parameters and type parameters (cf p.8). Variable parameters are considered local variables to the unit. A variable parameter has one of three transmission modes: input, output or inout. If no mode is explicitly given the input mode is assumed. For instance in the unit declaration:

```
unit P: procedure(x,y:real,b:boolean;
           output c:char,i:integer;inout :integer);
```

x,y,b are input parameters , c,i are output parameters , and j is inout parameter.

Input parameter acts as a local variable whose value is initialized by the value of the corresponding actual parameter. Output parameter acts as a local variable initialized in the standard manner (real with 0.0, integer with 0, boolean with false etc.). Upon return its value is assigned to the corresponding actual parameter, in which case it must be a variable. However the address of such an actual parameter is determined upon entry to the body. Inout parameter acts as an input parameter and output parameter together.

Example:

```
unit squareeq: procedure(a,b,c:real;output xr,xi,yr,yi:real);

 (* given a,b,c the procedure solves  square equation :
    ax*x+bx+c=0.
     xr,xi- real and imaginary part of the first root
     yr,yi- real and imaginary part of the second root *)
var delta: real;

begin     (*a=/=0*)

  a:=2*a; c:=2*c; delta:=b*b-a*c;
  if delta <= 0

  then

    xr,yr:=-b/a;
    if delta=0 then  return fi;     (*xi=yi=0 by default*)
    delta:=sqrt(-delta);
    xi:=delta/a; yi:=-xi;
    return


  fi;


  delta:=sqrt(delta);
  if b=0


  then


    xr:=delta/a; yr:=-xr;
    return


  fi;


  if b>0 then b:=b+delta else b:=b-delta fi;
  xr:=-b/a; yr:=-c/b;
end squareeq;
```

A procedure call to the above unit may be the following:

```
call squareeq(3.75*H,b+7,3.14,g,gi,h,hi);
```

where g,h,gi,hi are real variables.

No restriction is imposed on the order of declarations. In particular, recursive procedures and functions can be declared without additional announcements (in contrast to Pascal).

Example:

For two recursive sequences defined as:

```
 a(n)=b(n-1)+n+2        n>0
b(n)=a(n-1)+(n-1)*n     n>0
a(0)=b(0)=0
```

one can declare two functions:

```
unit a: function(n:integer):integer;
begin
```

```
  if n>0 then result:=b(n-1)+n+2 fi
end a;

unit b: function(n:integer):integer;
begin

  if n>0 then result:=a(n-1)+(n-1)*n fi

end b;
```

and invoke them:

```
k:=a(100)*b(50)+a(15);
```

Functions and procedures can be formal parameters as well.

Example:

```
unit Bisec: procedure(a,b,eps:real;output x:real;function
                                        f(x:real):real);
(*this procedures searches for zero of the continous function f in
                                        the segment (a,b) *)
var h:real,s:integer;
begin
  s:=sign(f(a));
  if sign(f(b))=s then return fi;   (* wrong segment *)

  h:=b-a;
  do

    h:=h/2; x:=a+h;
    if h < eps then  return fi;
    if sign(f(x))=s then a:=x else b:=x fi
  od

end Bisec;
```

In the above declaration, after the input variable parameters a,b,eps and the output variable parameter x, a function parameter f appears. Note that its specification part is complete. Thus the check of actual-formal parameter compatibility is possible at compilation time. Making use of this syntactic facility is not possible in general, if a formal procedure (function) is again a formal parameter of a formal procedure (function). The second degree of formal procedures (functions) nesting is rather scarce, but LOGLAN-82 admits such a construct. Then formal procedure (function) has no specification part and the full check of actual-formal parameter compatibility is left to be done at run time.

Example:

```
unit P: procedure(j:integer; procedure G (i:integer;
                                 procedure H));
  ...
begin


  ...
  call G(j,P);
end P;
```

Procedure G is a first degree parameter, therefore it occurs with complete specification part. Procedure H is a second degree parameter and has no specification part. In this case a procedure call can be strongly recursive:

```
  call P(i+10,P);
```

4. Classes

Class is a facility which covers such programming constructs as structured type, package, access type, data structure etc. To begin with the presentation of this construct, let us consider a structured type assembled from primitive ones:

```
unit bill: class;
  var  dollars          :real,
       not_paid         :boolean,
       year,month,day    :integer;
end bill;
```

The above class declaration has the attributes : dollars (real), not_paid (boolean), and year,month,day (integer). Wherever class bill is visibile one can declare variables of type bill:

```
var x,y,z: bill;
```

The values of variables x, y, z can be the addresses of objects of class bill. These variables are called reference variables. With reference variable one can create and operate the objects of reference variable type.

An object of a class is created by the class generation statement (new), and thereafter, its attributes are accessed through dot
notation.

```
x:=new bill; (* a new object of class bill is created *)
x.dollars:=500.5;  (* define amount *)
x.year:=1982;      (* define year *)
x.month:=3;        (* define month *)
x.day:=8;          (* define day *)
y:=new bill;       (* create a new object *)


y.not_paid:=true;  (* bill not_paid *)
z:=y;       (* variable z points the same object as y *)
```

If an object of class bill has been created (new bill) and its
address has been assigned to variable x (x:=new bill), then the
attributes of that object are accessible through dot notation (remote access). The expression x.dollars gives , for instance, the remote access to attribute dollars of the object referenced by x. All attributes of class objects are initialized as usual. For the above example the object referenced by x, after the execution of the specified sequence of statements, has the following structure:

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄż
ł   500.5   ł    dollars
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł   false   ł    not_paid
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł   1982    ł    year
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł     3     ł    month
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł     8     ł    day
ŔÄÄÄÄÄÄÄÄÄÄÄÄÄÄŮ
```

The object referenced by y and z has the following structure:

```
ÚÄÄÄÄÄÄÄÄÄÄÄÄÄż
ł    0    ł    dollars
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł   true   ł    not_paid
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł    0    ł    year
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł    0    ł    month
ĂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
ł    0    ł    day
ŔÄÄÄÄÄÄÄÄÄÄÄÄÄŮ
```

The value none is the default initial value of any reference
variable and denotes no object. A remote access to an attribute of none raises a run time error.

Class may have also formal parameters (as procedures and functions). Kinds and transmission modes of formal parameters are the same as in the case of procedures.

Example:

```
unit node: class (a:integer);
 var left,right:node;


end node;
```

Let, for instance, variables z1, z2, z3 be of type node. Then the sequence of statements:

```
z1:=new node(5);
z2:=new node(3);


z3:=new node(7);


z1.left:=z2; z1.right:=z3;
```

creates the structure:

```
          ÚÄÄÄÄÄÄÄÄÄÄż
      z1ÄÄÄÄÄÄ´   5   ł
          ĂÄÄÄÄÄÄÄÄÄÄ´
       ÚÄÄÄÄÄÄ´  left  ł
       ł     ĂÄÄÄÄÄÄÄÄÄÄ´
       ł     ł  right ĂÄÄÄÄÄÄÄÄÄż
       ł     ŔÄÄÄÄÄÄÄÄÄÄŮ      ł
       ł                 ł
    ÚÄÄÄÄÄÁÄÄÄÄÄż          ÚÄÄÄÄÄÁÄÄÄÄÄż
z2ÄÄÄÄÄ´   3   ł           ł   7   ĂÄÄÄÄÄÄÄz3
    ĂÄÄÄÄÄÄÄÄÄÄÄ´          ĂÄÄÄÄÄÄÄÄÄÄÄ´
    ł  none  ł           ł  none  ł
    ĂÄÄÄÄÄÄÄÄÄÄÄ´          ĂÄÄÄÄÄÄÄÄÄÄÄ´
    ł  none  ł           ł  none  ł
    ŔÄÄÄÄÄÄÄÄÄÄÄŮ          ŔÄÄÄÄÄÄÄÄÄÄÄŮ
```

where arrows denote the values of the reference variables.

Class may also have a sequence of statements (as any other unit). That sequence can initialize the attributes of the class objects.

Example:

```
unit complex:class(re,im:real);

var module:real;

begin

  module:=sqrt(re*re+im*im)
end complex;
```

Attribute module is evaluated for any object generation of class complex:

```
z1:=new complex(0,1); (* z1.module equals 1 *)
z2:=new complex(2,0); (* z2.module equals 2 *)
```

For the execution of a class generator, first a class object is created, then the input parameters are transmitted , and finally, the sequence of statements (if any) is performed. Return is made with the execution of return statement or the final end of a unit. Upon return the output parameters are transmitted.

Procedure object is automatically deallocated when return is made to the caller. Class object is not deallocated , its address can be assigned to a reference variable, and its attributes can be thereafter accessed via this variable.

The classes presented so far had only variable attributes. In general, class attributes may be also other syntactic entities, such as constants, procedures, functions, classes etc. Classes with procedure and function attributes provide a good facility to define data structures.

Example:

A push_down memory of integers may be implemented in the following way:

```
unit push_down :class;

 unit elem:class(value:integer,next:elem);
  (* elem - stack element *)
 end elem;

 var top:elem;

 unit pop: function :integer;

 begin

   if top=/= none

   then

     result:=top.value; top:=top.next
   fi;

 end pop;

 unit push:procedure(x:integer); (* x - pushed integer *)
 begin

   top:=new elem(x,top);
 end push;
```

```
        end push_down;
```

Assume that somewhere in a program reference variables of type push_down are declared (of course, in place where push_down is visibile):

```
        var s,t,z:push_down;
```

Three different push_down memories may be now generated:

```
        s:=new push_down(100); t:=new push_down(911); z:=new push_down(5);
```

One can use these push_down memories as follows:

```
        call s.push(7); (* push  7 to s *)

        call t.push(1); (* push  1 to t *)

        i:=z.pop;       (* pop an element from z *)
```

etc.

5. Adjustable arrays

In LOGLAN-82 arrays are adjustable at run time. They may be treated as objects of specified standard type with index instead of identifier selecting an attribute. An adjustable array should be declare somewhere among the lists of declarations and then may be generated in the sequence of statements.

Example:

```
       block

        var n,j:integer;

        var A:arrayof integer;  (* here is the declaration of A *)
```

```
begin


 read(n);
 array A dim (1:n);   (* here is the generation of A *)


 for i:=1 to n


 do


  read(A(i));
 od;


 (* etc.*)
end
```

A variable A is an array variable. Its value should be the reference to an integer array, i.e. a composite object consisting of integer components each one defined by an integer index.
Array generation statement:

```
array A dim (1:n);
```

allocates a one-dimensional integer array with the index bounds 1,n , and assigns its address to variable A.
The figure below illustrates this situation:

```
      ÚÄÄÄÄÄÄÄÄÄÄż            ÚÄÄÄÄÄÄÄÄÄÄÄż
      ł      ł           ł A(1)  ł      ł      ł           ĂÄÄÄÄÄÄÄÄÄÄÄ´
      ł  ...  ł           ł A(2)  ł      ĂÄÄÄÄÄÄÄÄÄÄ´            ĂÄÄÄÄÄÄÄÄÄÄÄ´
      ł   n  ł           ł       ł      ĂÄÄÄÄÄÄÄÄÄÄ´           ł  ...  ł      ł   j  ł
ł       ł       ĂÄÄÄÄÄÄÄÄÄÄ´            ĂÄÄÄÄÄÄÄÄÄÄÄ´
      ł   A   ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´   A(n)  ł       ŔÄÄÄÄÄÄÄÄÄÄŮ
ŔÄÄÄÄÄÄÄÄÄÄÄŮ
        Block object             Array object
```

A general case of array generation statement has the form:

array A **dim** (lower:upper)

where lower and upper are arithmetic expressions which define the range of the array index.

Example:

Two-dimensional array declaration :

```
        **var** A: **arrayof arrayof** integer;
```

and generation:

```
        **array** A **dim** (1:n)
        **for** i:=1 **to** n **do array** A(i) **dim** (1:m) **od**;
```

create the structure:

```
                         ÚÄÄÄÄÄÄÄÄÄż
                         ł A(1,1) ł
                         ĂÄÄÄÄÄÄÄÄÄ´
                         ł      ł
                         ł  ...  ł
                         ł      ł
ÚÄÄÄÄÄÄÄÄÄÄÄÄż              ĂÄÄÄÄÄÄÄÄÄł
ł  A(1)  ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´ A(1,m) ł
łÄÄÄÄÄÄÄÄÄÄÄ´              ŔÄÄÄÄÄÄÄÄÄŮ
ł        ł
ł   ...  ł
ł        ł
ĂÄÄÄÄÄÄÄÄÄÄÄÄ´              ÚÄÄÄÄÄÄÄÄÄż
ł  A(n)  ĂÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´ A(n,1) ł
ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ              ĂÄÄÄÄÄÄÄÄÄ´
                         ł      ł
                         ł  ...  ł
                         ł      ł
                         ĂÄÄÄÄÄÄÄÄÄ´
                         ł A(n,m) ł
                         ŔÄÄÄÄÄÄÄÄÄŮ
```

```
        **block**

          **var** i,j:integer, A,B: **arrayof arrayof** real, n:integer;
        **begin**
```

```
   read(n);
   array A dim (1:n);
   for i:=1 to n do array A(i) dim (1:n) od;


    (* A is square array *)
   array B dim (1:n);


   for i:=1 to n do array B(i) dim(1:i) od;
    (* B is lower triangular array *)
   A(n,n):=B(n,n);
   B(1):=A(1);
   B(1):=copy(A(1));
  end
```

Array A is the square array n by n. Each element A(i) , 1≤i≤n contains the address of row A(i,j), 1≤j≤n. Array B is the lower-triangular array. Each element B(i), 1≤i≤n, contains the address of row B(i,j), 1≤j≤i. Thus an assignment statement A(n,n):=B(n,n) transmits real value B(n,n) to real variable A(n,n). Assignment B(1):=A(1) transmits the address of the first row of A to variable B(1). Finally assignment B(1):=copy (A(1)) creates a copy of
the first row of A and assigns its address to B(1).

Upper and lower bounds of an adjustable array A are determined by standard operators lower(A) and upper(A).

Example:

```
  unit sort: procedure(A:arrayof integer);
   (*  insertion sort *)
   var n,i,j:integer; var x:integer;
  begin


   n:=upper(A);              (* assume lower bound is 1 *)
   for i:=2 to n


   do


     x:=A(i); j:=i-1;
     do


       if x >= A(j) then exit fi;
       A(j+1):=A(j);  j:=j-1;
       **if** j=0 **then exit fi**;
     **od**;


     A(j+1):=x
   **od**;


  **end** sort;
```

If an array variable A refers to no array its value is equal none (the standard default value of any array variable). An attempt to access an array element (e.g. A(i)) or a bound (e.g. lower(A)), where A is none, raises a run time error.

6. Coroutines and semicoroutines

Coroutine is a generalization of class. A coroutine object is an object such that the execution of its sequence of statements can be suspended and reactivated in a programmed manner. Consider first a simple class with a sequence of statements such that after return some non-executed statements remain. The generation of its object terminates with the execution of return statement, although the object can be later reactivated. If such a class is declared as a coroutine, then its objects may be reactivated. This can be realized by attach statement:

```
attach(X)
```

where X is a reference variable designating the activating coroutine object.

In general, since the moment of generation a coroutine object is either active or suspended. Any reactivation of a suspended coroutine object X (by attach(X)) causes the active coroutine object to be suspended and continues the execution of X from the statement following the last executed one.

Main program is also a coroutine. It is accessed through the standard variable main and may be reactivated (if suspended) by the statement attach(main).

Example:

In the example below the cooperation of two coroutines is presented. One reads the real values from an input device, another prints these values in columns on a line-printer, n numbers in a line. The input stream ends with 0.

```
program prodcons;
 var prod:producer,cons:consumer,n:integer,mag:real,last:bool;
 unit producer: coroutine;
 begin

  return;

  do

   read(mag);  (* mag- nonlocal variable, common store *)
   if mag=0

   then          (* end of data *)
    last:=true;
    exit

   fi;

   attach(cons);

  od;

  attach(cons)

 end producer;



 unit consumer: coroutine(n:integer);
 var Buf:arrayof real;
 var i,j:integer;

 begin
```

```
    array Buf dim(1:n);
    return;

    do

      for i:=1 to n

      do

        Buf(i):=mag;
        attach(prod);

        if last then exit exit fi;
      od;

      for i:=1 to n

      do     (* print Buf *)

        write(' ',Buf(i):10:2)
      od;

      writeln;
    od;

    (* print the rest of Buf *)
    for j:=1 to i do write(' ',Buf(j):10:2) od;

    writeln;
    attach(main);

  end consumer;


begin

    prod:=new producer;
```

```
      read(n);
      cons:=new consumer(n);


      attach(prod);


      writeln;
    end prodcons;
```

The above task could be programmed without coroutines at all. The presented solution is, however, strictly modular, i.e. one unit realizes the input process, another realizes the output process, and both are ready to cooperate with each other.

LOGLAN-82 provides also a facility for the semi-coroutine operations. This is gained by the simple statement detach. If X is the active coroutine object, then detach reactivates that coroutine object
at where the last attach(X) was executed. This statement meets the
need for the asymetric coroutine cooperations. (by so it is called semi-coroutine operation). Operation attach requires a reactivated coroutine to be defined explicitly by the user as an actual parameter. Operation detach corresponds in some manner to return in procedures. It gives the control back to a coroutine object where the last attach(X) was executed, and that coroutine object need not be known explicitly in X. This mechanism is, however, not so secure as the normal control transfers during procedure calls and returns.

In fact, the user is able to loop two coroutines traces by :

```
  attach(Y) in X
     attach(X) in Y
```

Then **detach** in X reactivates Y, **detach** in Y reactivates X.

In the example below the application of detach statement is illustrated.

Example:

```
    program reader_writers;
    (* In this example a single input stream consisting of blocks of numbers, each
    ending with 0, is printed on two printers of different width. The choice of the
    printer is determined by the block header which indicates the desired number of
    print columns. The input stream ends with a double 0. m1 - the width of
    printer_1, m2 - the width of printer_2 *)
     const m1=10,m2=20;


     var reader:reading,printer_1,printer_2:writing;
```

```
var n:integer,new_sequence:boolean,mag:real;

  unit writing:coroutine(n:integer);

    var Buf: arrayof real, i,j:integer;

  begin

    array Buf dim (1:n);      (* array  generation *)

    return;(* return terminates coroutine initialization *)

    do

      attach(reader);   (* reactivates coroutine reader *)
      if new_sequence

      then
    (* a new sequence causes buffer Buf to be cleared up *)
        for j:=1 to i do write(' ',Buf(j):10:2) od;
        writeln;
        i:=0; new_sequence:=false;  attach(main)

      else

        i:=i+1;   Buf(i):=mag;
        if i=n

        then

          for j:=1 to n do write(' ',Buf(j):10:2) od;
          writeln;
          i:=0;
        fi

      fi
    od

  end writing;



  unit reading: coroutine;

  begin

    return;

    do

      read(mag);
      if mag=0  then  new_sequence:=true;   fi;

      detach;
        (* detach returns control to printer_1 or printer_2
                depending which one reactivated the reader *)
    od

  end reading;



  begin

    reader:=new reading;

    printer_1:=new writing(m1); printer_2:=new writing(m2);
    do

      read(n);
      case n

        when 0:  exit

        when m1: attach(printer_1)
```

```
         when m2: attach(printer_2)

         otherwise  write(" wrong data"); exit

        esac

      od

    end;
```

Coroutines play the substantial role in process simulation. Class Simulation provided in Simula-67 makes use of coroutines at most degree. LOGLAN-82 provides for easy simulation as well. The LOGLAN-82 class Simulation is implemented on a heap what gives lg(n) time cost (in contrast with O(n) cost of the original implementation). It covers also various simulation problems of large size and degree of complexity.

7. Prefixing

Classes and prefixing are ingenius inventions of Simula-67(cf [1]). Unfortunately they were hardly ever known and, perhaps, by this have not been introduced into many programming language that gained certain popularity. Moreover, implementation constraints of Simula-67 bind prefixing and classes workableness to such a degree that both facilities cannot be used in all respects. We hope that LOGLAN-82, adopting merits and rooting up deficiencies of these constructs, will smooth their variations and vivify theirs usefulness.

What is prefixing ? First of all it is a method for unit extending. Consider the simplest example:

```
    unit bill: class;

      var

     dollars          :real,
         not_paid          :boolean,
         year,month,day    :integer;
    end bill;
```

Assume the user desires to extend this class with new attributes. Instead of writing a completely new class, he may enlarge the existing one:

```
unit gas_bill:bill class;

  var cube_meters: real;

end gas_bill;
```

Class gas_bill is prefixed by class bill. This new declaration may appear anywhere within the scope of declaration of class bill. (In Simula-67 such a prefixing is forbidden in nested units.) Class gas_bill has all the attributes of class bill and additionally its own attributes (in this case the only one: cube_meters). The generation statement of this class has the form:

```
z:=new gas_bill;
```

where z is a reference variable of type gas_bill. Remote access to the attributes of prefixed class is standard:

```
z.dollars:=500.5; z.year:=1982; z.month:=3; z.day:=8;
z.cube_meters:=100000;
```

Consider now the example of a class with parameters.

Assume that in a program a class:

```
unit id_card: class(name:string,age:integer);

end id_card;
```

and its extension:

```
unit idf_card:id card class(first name:string);

end idf_card;
```

are declared.

Then for variable z of type id_card and variable t of type idf_card the corresponding generation statement may be the following:

```
z:=new id_card("kreczmar",37);

t:=new idf_card("Kreczmar",37,"Antoni");
```

Thus the formal parameters of a class are concatenated with the formal parameters of its prefix.

One can still extend class idf_card. For instance:

```
unit idr_card:idf_card class;

  var children_number:integer;

  var birth_place:string;

end idr_card;
```

Prefixing allows to build up hierarchies of classes. Each one hierarchy has a tree structure. A root of such a tree is a class without prefix. One class is a successor of another class iff the first is prefixed by the latter one.

Consider the prefix structure:

```
    A
   . . .
```

```
      . . .
     .  .  .
    B.   .C  .D
     .
      .
       .E
        .
         .
          .F
         . .
        .   .
       G.    .H
```

Class H has a prefix sequence A, B, E, F, H. Let a, b, ... , h denote the corresponding unique attributes of classes A, B, ... , H, respectively. The objects of these classes have the following forms:

```
    ÚÄÄÄÄÄÄÄÄÄÄÄż  ÚÄÄÄÄÄÄÄÄÄÄÄż  ÚÄÄÄÄÄÄÄÄÄÄÄż
ÚÄÄÄÄÄÄÄÄÄÄÄż
    ł   a   ł ł   a   ł ł   a   ł ł   a   ł
    ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´
ĂÄÄÄÄÄÄÄÄÄÄÄ´
     object A    ł   b   ł ł   c   ł ł   d   ł
                ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ
                 object B      object C      object D

    ÚÄÄÄÄÄÄÄÄÄÄÄż  ÚÄÄÄÄÄÄÄÄÄÄÄż  ÚÄÄÄÄÄÄÄÄÄÄÄż
ÚÄÄÄÄÄÄÄÄÄÄÄż
    ł   a   ł ł   a   ł ł   a   ł ł   a   ł
    ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´
ĂÄÄÄÄÄÄÄÄÄÄÄ´
    ł   b   ł ł   b   ł ł   b   ł ł   b   ł
    ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´
ĂÄÄÄÄÄÄÄÄÄÄÄ´
    ł   e   ł ł   e   ł ł   e   ł ł   e   ł
    ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  łÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´
ĂÄÄÄÄÄÄÄÄÄÄÄ´
     object E    ł   f   ł ł   f   ł ł   f   ł
                ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  ĂÄÄÄÄÄÄÄÄÄÄÄ´  ĂÄÄÄÄÄÄÄÄÄÄÄ´
                 object F   ł   g   ł ł   h   ł
                            ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ  ŔÄÄÄÄÄÄÄÄÄÄÄÄŮ
                                                  object G      object H
```

Let Ra, Rb,..., Rh denote reference variables of types A, B,..., H, respectively. Then the following expressions are correct:

Ra.a, Rb.b, Rb.a, Rg.g, Rg.f, Rh.h, Rh.f, Rh.e, Rh.b, Rh.a etc.

Variable Ra may designate the object of class B (or C,..., H), i.e. the statement:

```
Ra:=new B
```

is legal. But then attribute b is not accessible through dot via Ra, i.e. Ra.b is incorrect. This follows from insecurity of such a remote access. In fact, variable Ra may point any object of a class prefixed by A, in particular, Ra may point the object of A itself, which has no attribute b. If Ra.b had been correct, a compiler should have distiguish the cases Ra points to the object of A or not. But this, of course, is undistinguishable at compilation time.

To allow, however, the user's access to attribute b (after instruction Ra:=**new** B), the instantaneous type modification is provided within the language:

```
Ra qua B
```

The correctness of this expression is checked at run time. If Ra designates an object of B or prefixed ba B, the type of the expression is B. Otherwise the expression is erroneous. Thus, for instance, the expressions:

```
Ra qua G.b,    Ra qua G.e    etc.
```

enable remote access to the attributes b, c, ... via Ra.

So far the question of attribute concatenation was merely discussed. However the sequences of statements can be also concatenated.

Consider class B prefixed with class A. In the sequence of statements of class A the keyword inner may occur anywhere, but only once. The sequence of statements of class B consists of the sequence of statements of class A with inner replaced by the sequence of
statements of class B.

```
unit A :class              unit B:A class

   ...                        ...
begin                      begin

   ...                   ÚÄÄÄ...
```

```
                              ł                                inner
ÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´ inner

                              ł
     ...                      ŔÄÄÄ...
   end A;                         end B;
```

In this case inner in class B is equivalent to the empty statement.
If class B prefixes another class, say C, then inner in B is replaced
by the sequence of statements of class C, and so on. If inner does not occur explicitly, an implicit occurrence of inner
before the final end of a class is assumed.

Example

Let class complex be declared as usual:

```
        unit complex: class(re,im:real);


        end complex;
```

and assume one desires to declare a class mcomplex with the additional attribute module. In order the generation of class mcomplex define the value of attribute module, one can declare a class:

```
        unit mcomplex:complex class;


        var module:real;


        begin


          module:=sqrt(re*re+im*im)
        end mcomplex;
```

Class mcomplex may be still extended:

```
unit pcomplex:mcomplex class;

  var alfa:real;

begin

  alfa:=arccos(re/module)
end pcomplex;
```

For these declarations each generation of class mcomplex defines the value of attribute module, each generation of class pcomplex defines the values of attributes module and alfa.

For reference variables z1, z2 z3 of type complex, the following sequence of statements illustrates the presented constructs:

```
z1:=new complex(0,1);

z2:=new mcomplex(4,7);

z3:=new pcomplex(-10,12);

if z2 qua mcomplex.module > 1

then

    z1:=z2;
fi;

if z3 qua pcomplex.alfa < 3.14

then

   z3.re:=-z3.re;  z3.alfa:=z3.alfa+3.14;
fi;
```

```
z1 qua mcomplex.module:= 0;


z1.re,z1.im:=0;
```

Example:

Binary search tree (Bst) is a binary tree where for each node x the nodes in the left subtree are less than x, the nodes in the right subtree are greater than x. It is the well-known exercise to program the algorithms for the following operations on Bst:

member(x) = true iff x belongs to Bst
insert(x), enlarge Bst with x, if x does not yet belong to Bst

We define both these operations in a class:

```
unit Bst: class;

  unit node: class(value:integer);  (*  tree node  *)

    var left,right:node;

  end node;

  var root:node;

  unit help: class(x:integer);      (* auxiliary class *)

    var p,q:node;

  begin

     q:=root;
     while q=/= none

     do

       if x < q.value

       then
```

```
          p:=q; q:=q.left;
          **repeat**  (* jump to the beginning of a loop *)

        **fi**;

        **if** q.value < x

        **then**

          p:=q; q:=q.right;  **repeat**

        **fi**;

        **exit**

      **od**;

      **inner**
      (* virtual instruction to be˙replaced

by the body of
        a module prefixed by help  *)
    **end** help;

    **unit** member:help **function**:boolean;

  (* x is a formal parameter derived from the prefix help *)
    **begin**

      **result**:=q=/=none

    **end** member;

    **unit** insert:help **procedure**;

  (* x is a formal parameter derived from the prefix help *)
    **begin**

      **if** q=/=none **then return fi**;
      q:=new node(x);

      if p=none then root:=q; return fi;

      if p.value < x then p.right:=q else p.left:=q fi;

    end insert;

  begin

    inner;

  end Bst;
```

In the example the common actions of member and insert are programmed in class help. Then it suffices to use class help as a prefix of function member and procedure insert, instead of redundant occurrences of the corresponding sequence of statements in both units.

Class Bst may be applied as follows:

```
var X,Y:Bst;

begin

    X:=new Bst;  Y:=new Bst;

    call X.insert(5);

    if Y.member(-17) then ....

end
```

As shown in the declaration of Bst, class may prefix not only other classes but also procedures and functions. Class may prefix blocks as well.

Example:

Let class push_down (p. 5) prefix a block:

```
pref push_down(1000) block

var ...

begin

  ...
  call push(50); ...

  i:=pop;
  ...
end
```

In the above block prefixed with class push_down one can use pop and push as local attributes. (They are local since the block is embedded in the prefix push down.)

Example:

```
pref push down(1000) block

begin

  ...
  pref Bst block

  begin

  (* in this block both structures
      push down and Bst are visible *)
   call push(50);

   call insert(13);
```

```
    if member(10) then ...


    i:=pop;
    ...
  end


end
```

In place where classes push_down and Bst are visible together a block prefixed with Bst may be nested in a block prefixed with push_down (or vice versa). In the inner block both data structures are directly accessible. Note that this construct is illegal in Simula 67.

8. Formal types

Formal types serve for unit parametrization with respect to any non-primitive type.

Example:

```
unit Gsort:procedure(type T; A:arrayof T; function less
                               (x, y: T): boolean);
var n,i,j:integer; var x:T;


begin


  n:=upper(A);
  for i:=2 to n


  do


    x:=A(i); j:=i-1;
    do


      if less(A(j),x) then exit fi;


      A(j+1):=A(j); j:=j-1;
      if j=0 then exit fi;
    od;

    A(j+1):=x;
  od

end Gsort;
```

Procedure Gsort (the generalization of procedure sort from p.4) has type parameter T. A corresponding actual parameter may be an arbitrary non-primitive type. An actual parameter corresponding to A should be an array of elements of the actual type T. Function less should define the linear ordering on the domain T.

For instance, the array A of type bill (cf p.7) may be sorted with respect to attribute dollars , if the function:

```
unit less: function(t,u:bill):boolean;

begin

  result:=t.dollars <= u.dollars
end less;
```

is used as an actual parameter:

```
call Gsort(bill,A,less);
```

If the user desires to sort A with respect to date, it is sufficient to declare :

```
unit earlier:function(t,u:bill):boolean;

begin

  if t.year < u.year then result:= true; return  fi;

  if t.year=u.year

  then
```

```
    if t.month < u.month then result:=true; return fi;

    if t.month=u.month then result:=t.day<=u.day  fi

  fi;

 end earlier;
```

and to call: call Gsort(bill,A,earlier);

9. Protection techniques

Protection techniques ease secure programming. If a program is large, uses some system classes, is designed by a team etc., this is important (and non-trivial) to impose some restrictions on access to non-local attributes.

Let us consider a data structure declared as a class. Some of its attributes should be accessible for the class users, the others should not. For instance, in class Bst (p.7) the attributes member and insert are to be accessible. On the other hand the attributes root, node and help should not be accessible, even for a meddlesome user. An improper use of them may jeopardize the data structure invariants.

To forbid the access to some class attributes the three following protection mechanisms are provided:

close, hidden, and **taken.**

The protection close defined in a class forbids remote access to the specified attributes. For example, consider the class declaration:

```
unit A: class;

  close x,y,z;

  var  x: integer, y,z:real;

  ....
```

```
    end A
```

Remote access to the attributes x,y,z from outside of A is forbidden.

The protection hidden (with akin syntax) does not allow to use the
specified attributes form outside of A neither by the remote access nor in the units prefixed by A. The only way to use a hidden attribute is to use it within the body of class A.
Protection taken defines these attributes derived from prefix, which
the user wishes to use in the prefixed unit. Consider a unit B prefixed by a class A. In unit B one may specify the attributes of A which are used in B. This protects the user against an unconscious use of an attribute of class A in unit B (because of identifier conflict). When taken list does not occur, then by default, all non-hidden attributes of class A are accessible in unit B.

10. Programmed deallocation

The classical methods implemented to deallocate class objects are based on reference counters or garbage collection. Sometimes the both methods may be combined. A reference counter is a system attribute holding the number of references pointing to the given object. Hence any change of the value of a reference variable X is followed by a corresponding increase or decrease of the value of its reference counter. When the reference counter becomes equal 0, the object can be deallocated.

The deallocation of class objects may also occur during the process of garbage collection. During this process all unreferenced objects are found and removed (while memory may be compactified). In order to keep the garbage collector able to collect all the garbage, the user should clear all reference variables , i.e. set to None, whenever possible. This system has many disadvantages. First of all, the programmer is forced to clear all reference variables, even those which are of auxiliary character. Moreover, garbage collector is a very expensive mechanism and thus it can be used only in emergency cases.

In LOGLAN a dual operation to the object generator, the so-called object deallocator is provided. Its syntactic form is as follows:

```
            kill(X)
```

where X is a reference expression. If the value of X points to no object (none) then kill(X) is equivalent to an empty statement. If the
value of X points to an object O, then after the execution of kill(X),
the object O is deallocated. Moreover all reference variables which pointed to O are set

to none. This deallocator provides full *security*,
i.e. the attempt to access the deallocated object O is checked and results in a run-time error.

For example:

```
Y:=X;  kill(X);   Y.W:=Z;
```

causes the same run-time error as:

```
X:=none;  X.W:=Z;
```

The system of storage management is arranged in such a way that the frames of killed objects may be immediately reused without the necessity of calling the garbage collector, i.e. the relocation is performed automatically. There is nothing for it but to remember not to use remote access to a killed object. (Note that the same problem appears when remote access X.W is used and X=none).

Example:

Below a practical example of the programmed deallocation is presented. Consider class Bst (p.7). Let us define a procedure that deallocates the whole tree and is called with the termination of the class Bst.

```
unit Bst:class;

  (* standard declarations list of  Bst *)
 unit kill_all:procedure(p:node);

 (* procedure kill_all deallocates a tree with root p *)
 begin

   if p= none then return fi;

   call kill_all(p.left);
```

```
    call kill_all(p.right);

    kill(p)

   end kill_all;

   begin

    inner;

    call kill_all(root)

  end Bst;
```

Bst may be applied as a prefix:

```
  pref Bst block

   ...
  end
```

and automatically will cause the deallocation of the whole tree after return to call kill_all(root) from the prefixed block.

To use properly this structure by remote accessing one must call kill_all by himself:

```
  unit var X,Y:Bst;

   ...
  begin

    X:=new Bst;  Y:=new Bst;

      ...
    (* after the structures' application *)
    call X.kill_all(X.root);

    kill(X);

    call Y.kill_all(Y.root);

    kill(Y);

    ...
  end
```

Finally note that deallocator kill enables deallocation of array objects, and suspended coroutines and processes as well (cf p.13).

11. Exception handling

Exceptions are events that cause interruption of normal program execution. One kind of exceptions are those which are raised as a result of some run time errors. For instance, when an attempt is made to access a killed object, when the result of numeric operation does not lie within the range, when the dynamic storage allocated to a program is exceeded etc.

Another kind of exceptions are those which are raised explicitly by a user (with the execution of the raise statement).

The response to exceptions (one or more) is defined by an exception handler. A handler may appear at the end of declarations of any unit. The corresponding actions are defined as sequences of statements preceded by keyword when and an exception identifier.

Example:

In procedure squareeq (p.3) we wish to include the case when a=0. It may be treated as an exception (division by zero).

```
  unit squareeq(a,b,c:real;output xr,xi,yr,yi:real);

    var delta:real;
```

```
   handlers
    when division_by_zero:
    if b =/= 0
    then
      xi,yr,yi:=0; xr:=-c/b; terminate
    else
      raise Wrong_data(" no roots")
    fi;
end
begin
  ...
end squareeq;
```

The handler declared in that procedure handles the only one exception (division_by_zero).

When an exception is raised, the corresponding handler is searched for, starting from the active object and going through return traces. If there is no object containing the declaration of the handler, then the program (or the corresponding process) is terminated. Otherwise the control is transferred to the first found handler.

In our example the handler is declared within the unit itself, so control is passed to a sequence:

```
if b=/=0
  ...
```

Therefore, when b=/=0, the unique root of square equation will be determined and the procedure will be normally terminated (terminate).

In general, terminate causes that all the objects are terminated, starting from that one where the exception was raised and ending on that one where the handler was found. Then the computation is continued in a normal way.

In our example, when b=0, a new exception is raised by the user. For this kind of exceptions , the exception itself should be declared (because it is not predefined as a run time error). Its declaration may have parameters which are transmitted to a handler. The exception declaration need not be visible by the exception handler. However the way the handler is searched for does not differ from the standard one. Consider an example:

```
block
 signal Wrong_data(t:string);


 unit squareeq:
      ...
 end squareeq;
 ...
begin


    ...
end
```

Exception Wrong_data may be raised wherever its declaration (signal Wrong_data) is visible. When its handler is found the specified sequence of actions is performed. In the example above different handlers may be defined in inner units to the main block where squereeq is called.

The case a=0 could be included, of course, in a normal way, i.e. by a corresponding conditional statement occurring in the procedure body. But the case a=0 was assumed to be exceptional (happens scarcely). Thus the evaluation of condition a=0 would be mostly unnecessary. As can be noticed thanks to exceptions the above problem can be solved with the minimal waste of run time.

12. Concurrent processes.

Loglan allows to create and execute objects-processes. They can operate simultaneously on different computers linked into a LAN network or a few processes can share one processor (its time-slices).

Process modules are different from the classes and coroutines for, they use the keyword **process**. The syntax of process modules is otherwise the same. In a process one can use a few more instructions: resume (resume a process which is passive), stop - make the current process passive, etc.

All processes (even those executed on the same computer) are implemented as distributed, i.e. without any shared memory. This fact implies some restrictions on how processes may be used. Not all restrictions are enforced by the present compiler, so it is the programmer's responsibility to respect them. For the details see the User's Manual.

Semantics of the generator **new** is slightly modified when applied to the processes. The first parameter of the first process unit in the prefix sequence must be of type INTEGER. This parameter denotes the node number of the computer on which this process will be created. For a single computer operation this parameter must be equal to 0.

Example:

```
unit A:class(msg:string);
...
end A;
unit P:A process(node:integer, pi:real);
...
end P;
...
var x:P;
...
begin
...
 (* Create process on node  4.  The  first  parameter  is  the  *)
 (*string required by the prefix A, the second is the node number *)
 x := new P("Hello", 4, 3.141592653);
...
end
```

COMMUNICATION MECHANISM

Processes may communicate and synchronize by a mechanism based on rendez-vous. It will be referred to as "alien call" in the following description.

An alien call is either:
- a procedure call performed by a remote access to a process object, or
- a call of a procedure which is a formal parameter of a process, or
- a call of a procedure which is a formal parameter of an alien-called procedure (this is a recursive definition).

Every process object has an enable mask. It is defined as a subset of all procedures declared directly inside a process unit or any unit from its prefix sequence (i.e. subset of all procedures that may be alien-called).

A procedure is enabled in a process if it belongs to that process' enable mask. A procedure is disabled if it does not belong to the enable mask.

Immediately after generation of a process object its enable mask is empty (all procedures are disabled).

Semantics of the alien call is different from the remote call described in the report. Both the calling process and the process in which the procedure is declared (i.e. the called process) are involved in the alien call. This way the alien call may be used as a synchronization mechanism.

The calling process passes the input parameters and waits for the call to be completed.

The alien-called procedure is executed by the called process. Execution of the procedure will not begin before certain conditions are satisfied. First, the called process must not be suspended in any way. The only exception is that it may be waiting during the ACCEPT statement (see below). Second, the procedure must be enabled in the called process.

When the above two conditions are met the called process is interrupted and forced to execute the alien-called procedure (with parameters passed by the calling process).

Upon entry to the alien-called procedure all procedures become disabled in the called process.

Upon exit the enable mask of the called process is restored to that from before the call (regardless of how it has been changed during the execution of the procedure). The called process is resumed at the point of the interruption. The execution of the ACCEPT statement is ended if the called process was waiting during the ACCEPT (see below). At last the calling process reads back the output parameters and resumes its execution after the call statement.

The process executing an alien-called procedure can easily be interrupted by another alien call if the enable mask is changed.

There are some new language constructs associated with the alien call mechanism. The following statements change the enable mask of a process:

```
        ENABLE p1, ..., pn
```

enables the procedures with identifiers p1, ..., pn. If there are any processes waiting for an alien call of one of these procedures, one of them is chosen and its request is processed. The scheduling is done on a FIFO basis, so it is strongly fair. The statement:

```
   DISABLE p1, ..., pn
```

disables the procedures with identifiers p1, ..., pn.

In addition a special form of the RETURN statement:

```
   RETURN ENABLE p1, ..., pn DISABLE q1, ..., qn
```

allows to enable the procedures p1, ..., pn and disable the procedures q1,...,qn after the enable mask is restored on exit from the alien-called procedure. It is legal only in the alien-called procedures (the legality is not enforced by the compiler).

A called process may avoid busy waiting for an alien call by means of the ACCEPT statement:

ACCEPT p1, ..., pn

adds the procedures p1, ..., pn to the current mask, and waits for an alien call of one of the currently enabled procedures. After the procedure return the enable mask is restored to that from before the ACCEPT statement.

Note that the ACCEPT statement alone (i.e. without any ENABLE/DISABLE statements or options) provides a sufficient communication mechanism. In this case the called process may execute the alien-called procedure only during the ACCEPT statement (because otherwise all procedures are disabled). It means that the enable mask may be forgotten altogether and the alien call may be used as a pure totally synchronous rendez-vous. Other constructs are introduced to make partially asynchronous communication patterns possible.

Below find a complete listing of a simple example - monitors.

```
program monitors;

(* this an example showing 5 processes: two of them are in fact monitors, one
controls the screen=ekran *)

  unit ANSI: class;
  (* CHECK whether config.sys contains a line
       device=ansi.sys
     the class ANSI enables operations on cursor,
                   and bold, blink, underscore etc. *)

  unit Bold : procedure;
  begin
    write( chr(27), "[1m")
  end Bold;

  unit Blink : procedure;
  begin
    write( chr(27), "[5m")
  end Blink;

  unit Reverse : procedure;
  begin
    write( chr(27), "[7m")
  end Reverse;

  unit Normal : procedure;
  begin
    write( chr(27), "[0m")
  end Normal;

  unit Underscore : procedure;
  begin
    write( chr(27), "[4m")
  end Underscore;

  unit inchar : IIUWgraph function : integer;
    (*podaj nr znaku przeslanego z klawiatury *)
    var i : integer;
  begin
    do
      i := inkey;
      if i <> 0 then exit fi;
    od;
    result := i;
  end inchar;

  unit NewPage : procedure;
  begin
    write( chr(27), "[2J")
  end NewPage;

  unit  SetCursor : procedure(row, column : integer);
    var c,d,e,f  : char,
        i,j : integer;
  begin
    i := row div 10;
    j := row mod 10;
    c := chr(48+i);
    d := chr(48+j);
    i := column div 10;
    j := column mod 10;
    e := chr(48+i);
    f := chr(48+j);
    write( chr(27), "[", c, d, ";", e, f, "H")
  end SetCursor;
end ANSI;


    unit monitor:  process(node:integer, size:integer,e: ekran);

      var buf: arrayof integer,
          nr,i,j,k1,k2,n1,n2: integer;


    unit lire: procedure(output k: integer);
    begin
      call e.druk(13,2+nr*30+k1,0,k2);
      call e.druk(13,2+nr*30+(i-1)*6,1,buf(i));
```

```
     k1:=(i-1)*6;
     k:=buf(i);
     k2:=k;
     i:= (i mod size)+1;
     if i=j
     then
       call e.printtext("i equal j")
     fi;
   end lire;

   unit ecrire: procedure(n:integer);
   begin
     call e.druk(13,2+nr*30+n1,0,n2);
     call e.druk(13,2+nr*30+(j-1)*6,2,n);
     n1:=(j-1)*6;
     buf(j) := n;
     n2:=buf(j);
     j := (j mod size)+1;
     if i=j
     then
       call e.printtext("j equal i")
     fi;
   end ecrire;
 begin
   array buf dim(1:size);
   nr := size - 4;
   for i := 1 to size
   do
     buf(i) :=  i+nr*4;
     call e.druk(13,2+nr*30+(i-1)*6,0,buf(i));
   od;
   i:=1;
   j := size;
   k1:=0;
   k2:=buf(1);
   n1:=(size-1)*6;
   n2:=buf(size);
   (* end initialize buffer *)
   return;

   do
     accept lire, ecrire
   od
 end monitor;

 unit prcs:  process(node,nr:integer, mleft,mright:
                         monitor, e: ekran);
   var l,o: integer;

 begin
   call e.SetCursor(8+(nr-1)*10,29);
   if nr = 1
```

```
      then
        call e.printtext("<-- p1 <--");
      else
        call e.printtext("--> p2 -->");
      fi;
      return;
      do
        call mleft.lire(l) ;
        call e.druk(11+(nr-1)*4,31-(nr-1)*8,1,l);
        l:= l+1;
        call mright.ecrire(l) ;
        call e.druk(10+(nr-1)*6,23+(nr-1)*8,2,l);
        if l mod 15 = 0
        then
          o:= e.inchar;
            if o = -79 then call endrun fi;
        fi;
      od;
    end prcs;


  unit ekran : ANSI process(nrprocesora: integer);
      unit printtext: procedure(s:string);
      begin
        write(s);
        call Normal;
      end printtext;

      unit  druk: procedure(gdzieW,gdzieK,jak,co:integer);
      begin
        call SetCursor(gdzieW,gdzieK);
        write("   ");
        if jak=0 then call Normal else
          if jak=1 then call Reverse else
            if jak=2 then call Bold
            fi
          fi
        fi;
        write(co:3);
        call Normal;
      end druk;

      unit print: procedure (i:integer);
      begin
        write(i:4)
      end print;
    begin
      return;

      do accept inchar,
              Normal,NewPage, SetCursor, Bold, Underscore,
           Reverse, Blink, print, printtext, druk
      od
    end ekran;

  var m1,m2:monitor,
      e:ekran,
      p1,p2:prcs;

  begin    (* ----- HERE IS THE MAIN PROGRAM ----- *)
    (* create a  configuration *)
    e:= new ekran(0);
    resume(e);
    call e.Normal;
    call e.NewPage;
    m1 := new monitor(0,4,e);
    m2 := new monitor(0,5,e);

    p1 := new prcs(0,1,m2,m1,e);
    p2 := new prcs(0,2,m1,m2,e);

    resume(m1);
    resume(m2);
    resume(p1);
    resume(p2);
  end monitors;
```

References.

Bartol,W.M., et al.
Report on the Loglan 82 programming Language,
Warszawa-Lodz, PWN, 1984

O.-J. Dahl, B. Myhrhaug, K. Nygaard,
Simula 67 Common Base Language,
Norwegian Computing Center, Oslo, 1970 the mother of object languages!!

Hoare C.A.R.
Monitors, an operating system structuring concept.
CACM,vol.17,N.10,October 1974,pp.549-57

Loglan'82
User's guide
Institute of Informatics, University of Warsaw 1983, 1988
LITA, Université de Pau, 1993
(distributed together with this package)

A.Kreczmar, A.Salwicki, M. Warpechowski,
Loglan'88 - Report on the Programming Language,
Lecture Notes on Computer Science vol. 414, Springer Vlg, 1990,
ISBN 3-540-52325-1

/* if you can read polish, there is a good manual of Loglan */
A.Szalas, J.Warpechowska,
LOGLAN,
Wydawnictwa Naukowo-Techniczne, Warszawa, 1991 ISBN 82-204-1295-1

see also the Readings file of this distribution.