# Examples

This is a set of examples as they were in Proceedings of Zaborów’83 Summer School

# Spis treści

# Merge

```
program Merge;
   (* COROUTINE MERGE OF BINARY TREES*)



unit NODE : class;
   (* NODE OF BINARY TREE *)
   var LEFT,RIGHT : NODE, VAL : INTEGER; (*SEARCHING KEY *)

   unit INS : procedure (VALUE : INTEGER);
     begin
       if VAL> VALUE
        then
         if LEFT = NONE
          then
           LEFT := NEW NODE;
           LEFT.VAL := VALUE

           else
            CALL LEFT.INS(VALUE)
           fi
         else
          (* ELEMENTS NOT LESS THAN VAL ARE LOCATED IN THE RIGHT SUBTREE*)
          if RIGHT = NONE
           then
            RIGHT := NEW NODE;
            RIGHT.VAL := VALUE
           else
            CALL RIGHT.INS(VALUE)
          fi
        fi;
   end INS;

end NODE;


   unit TRAVERS : COROUTINE (X :NODE);
   (* CONSECUTIVE ELEMENTS OF TREE NODE ARE LOCATED IN THE GROWING
ORDER to *)
   (* THE "MAIL BOX" VAL AND SENT to THE ATTACHING unit                    *)
      var VAL : INTEGER;

      unit T : procedure (Y : NODE);
      (* RECURSIVE procedure for INFIX TRAVERSION RESULTING TREE ELEMENTS  *)
      (* IN NOT DECREASING ORDER                                         *)
         begin
           if Y =/= NONE
            then
```

```
            CALL T(Y.LEFT);
            VAL := Y.VAL;
            DETACH;
            (* CONSECUTIVE ELEMENTS OF TREE Y ARE SENT for FURTHER        *)
            (* PROCESSING to THE MASTER PROGRAM                           *)
            CALL T(Y.RIGHT);
           fi
      end T;

      begin
        RETURN;
        CALL T(X);
        VAL := M;
        (* VAL IS MAXIMAL VALUE TREATED AS A SENTINEL while ENTIRE TREE IS *)
        (* TRAVESED                                                  *)
   end TRAVERS;

   var N,I,J,MIN,M,K : INTEGER,
    (* N - TNE NUMBER OF TREES
       M - MAXIMAL KEY VALUE + 1
       MIN- MINIMAL VALUE PRODUCED AT A GIVEN MOMENT BY SYSTEM OF
COROUTINES*)
       D : arrayof NODE,
       TR : arrayof TRAVERS;

begin
WRITELN(" PROGRAM USES COROUTINES AND MERGES A GIVEN NUMBER OF
BINARY",
        " SEARCHING TREES");
do WRITELN(" GIVE THE NUMBER OF TREES:");
   READ(N);
   WRITELN(N);
   if N>0 then EXIT else WRITELN(" THE NUMBER MUST BE > 0") fi
od;
WRITELN(" ELEMENTS OF THE TREES ARE INTEGERS");
WRITELN(" to TERMIATE INSERTING TREE TYPE  -1.");
WRITELN(" THIS NUMBER IS NOT INSERTED AS AN ELEMENT");

array D DIM(1:N);
for I := 1 to N do
   WRITELN(" GIVE THE ELEMENT SEQUENCE for THE TREE NO.",I:4);
   READ(J); WRITE(J);  if J>M then M :=J fi ;
   D(I) := NEW NODE;
   D(I).VAL := J;
   do
    READ(J);
     if J = -1 then WRITELN; EXIT fi;
     WRITE(J);
     if J > M then M := J fi;
     CALL D(I).INS(J)
   od;
od;
```

```
M := M+1;
WRITELN(" THE MERGED SEQUENCE IS:");

array TR DIM(1:N);

MIN := 0;
(* GENERATE THE TRAVERSERS SYSTEM *)
for I:= 1 to N do
   TR(I) := NEW TRAVERS (D(I));
   ATTACH(TR(I));
od;

K:=0;
do
  if MIN = M then EXIT fi;
  MIN := TR(1).VAL;
  J :=1;
  for I:= 2 to N do
   if MIN>TR(I).VAL then MIN:= TR(I).VAL; J := I fi;
  od;

if MIN< M then WRITE(' ',MIN); ATTACH(TR(J));
   K:=K+1; if K=10 then WRITELN fi
fi
od; WRITELN

end
```

# Treegen

```
program Treegen;
(* Generates the language defined by a  regular  expression*)
(* Program written by A. Kreczmar 1982
              proof   written by A. Salwicki    1990   *)


  unit REGEXP:coroutine;
          (* an object in the hierarchy of subtypes  of type REGEXP represents a regular expression
*)
(*
```

**Theorem**

For every object o in the hierarchy of classes that inherit from Regexp class the program *Pr* (see below), when executed will print all the words of the regular language represented by the object o and then it will stop.

```
Pr:  I:=0;
                    do
                      attach(o);
                      (* print the WORD *)
                                  for J:=1 to I
                                  do
                                    write(WORD(J))
                                  od;
                                  writeln;
                      if W.B then exit fi
                    od
```

**Lemma**

Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the **coroutine** o.
An execution of command **attach**(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the *new* word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

PROOF of the lemma is distributed in the subclasses of the class regexp, i.e. the proof goes by induction with respect to the length of a regular expression *)

```
    var B:BOOL; (* B ≡ all the words of the language were shown *)
  begin
    return
    inner;
    B := true
  end REGEXP;



   unit ATOM: REGEXP class(C:CHAR);
```

```
  (* an atomic regular expression consists of a letter
       Proposition. An execution of attach statement applied to this object will place the letter C on
I+1-th place
                            in the table WORD and the value of B will be assigned to true. In this way the
whole regular                          language is displayed at once.
      in this way we proved the base of the induction proof of the Lemma. *)
  begin
    do
      I:=I+1; (* update the position *)
      WORD(I):=C;
      B:=TRUE;
      detach
    od
  end ATOM;



  unit UNION: REGEXP class(L,R:REGEXP);
                (* represents the expression (L ∪ R) i.e. the union *)
(* Proposition.  Assume that objects L and R enjoy the property expressed by the Lemma
                 then any time this coroutine will be attached we obtain a subsequent word of the union of
the languages L and        R.

      Consider, a regular expresion of the length k. By our definition it is either a union object or a
concatenation object.
Let o be a union object i.e. o is UNION. The structure of its commands assures the following
            while not exhausted(L)
            do
                        attach(L)                    -- by induction hypothesis this command returns a word of L
language
            od
            (* L.B = true *)                              --  the exhaustion mark for L
            while not exhausted(R)
            do
                        attach(R)                    -- by induction hypothesis this command returns a word of R
language
            od
            (* R.B = true
                 B = true *)

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words
composed from the all words of L language followed by the sequence of all words from the R
language. *)

  var M: INTEGER;
  begin
    do    (* repeat : store I; generate one word (first from L next from R; detach; restore I until
exhausted *)
      M:=I;
            (* I is the position of the lastly generated letter.                *)
            (* M+1 is the position where the current UNION object   *)
            (* will place the letters of the currently generated word.    *)
```

```
    do
      attach(L);    (* by the inductive assumption this statement causes that one word will be
generated of the                                language L and it will be concatenated to the content of
WORD(1) , ... , WORD(I)  *)
      if L.B then exit fi;
      detach;
      I:=M  (* reestablish the position in the table WORD for the next word *)
    od;
    L.B:=FALSE;  (* restart language L *)
    do
      detach;
      I:=M;  (* reestablish the position in the table WORD for the next word  *)
      attach(R);    (* by the inductive assumption this statement causes that one word will be
generated of the                                language R and it will be concatenated to the content of
WORD(1) , ... , WORD(I)  *)
      if R.B then exit fi;
    od;
    R.B:=FALSE;  (* restart language R *)
    B:=TRUE;
    detach;
   od;
 end UNION;
```

```
 unit CONCATENATION: REGEXP class(L,R:REGEXP);
        (* represents the concatenation (L•R) of the languages represented by the regular
expressions L and R *)
      (* Suppose the object o is of the class CONCATENATION.

Now the loop of commands of object o assures basically the following
      while not exhausted
      do
        store (I);
        attach(L);        -- a word from L
        attach(R);        -- followed by a word from R
        detach;           -- hence a word of (L R) is given
        restore(I)
      od

with the necessary reactions to a case when one language (L or R) ends.
It is clear that if the object L and R enjoy the property mentioned in the Lemma then the object o
enjoys it too*)
```

```
var N,M:INTEGER;
begin
  do
    M:=I; (*M stores the begin position of first language word position *)
    do
      attach(L);
      N:=I; (* N stores the begin position of the second language word position *)
      do
        attach(R);
        if R.B then if L.B then exit exit else exit fi fi;
        detach; I:=N (* restart language R word generation position *)
      od;
      R.B:=FALSE; (* restart language R *)
      detach; I:=M (* restart language L word generation position *)
    od;
    R.B,L.B:=FALSE; B:=TRUE; detach
  od;
end CONCATENATION;



const N=50;   (* DIMENSION FOR ARRAY WORD *)

var    A,B,C,D,E,W,V,L,O,G,II,NN:REGEXP,
       I,J,N,M:INTEGER;
              (* I = GLOBAL POSITION POINTER FOR ARRAY WORD *)
var WORD: arrayof CHAR; (* BUFFER FOR WORDS GENERATION *)

begin
        writeln(" LANGUAGE GENERATOR USING COROUTINES");
        writeln(" LANGUAGE IS REPRESENTED AS A TREE WITH OPERATIONS IN
     NODES");
        writeln(" OUR OPERATIONS ARE SET THEORETICAL JOIN AND CONCATENATION
     OF");
        writeln(" LANGUAGES");writeln;
        A:=new ATOM('A'); B:=new ATOM('B'); C:=new ATOM('C');
        D:=new ATOM('D'); E:=new ATOM('E');
        L:=new ATOM('L'); G:=new ATOM('G');
        II:=new ATOM('I'); NN:=new ATOM('N');
        O:=new ATOM('O');
        W:=new UNION(A,L);
        W:=new CONCATENATION(W,new UNION(D,O));
        V:=new CONCATENATION(II,C);
        V:=new UNION(V,new CONCATENATION(L,new CONCATENATION(A,NN)));
        V:=new CONCATENATION(G,V);
        V:=new UNION(A,V);
        W:=new CONCATENATION(W,V);
        writeln(" WE HAVE LANGUAGE DEFINED BY THE FOLLOWING EXPRESSION");
        writeln;
        writeln(" (A∪L)•(D∪O)•(A∪G•(I•C∪L•A•N))");
```

```
   writeln; writeln;
   array WORD dim(1:N);
   do
      attach(W);
      write(" ");
      for J:=1 to I
      do
         write(WORD(J))
      od;
      writeln;
      if W.B then exit fi
   od
  end
```

(*

**Theorem**

For every object o in the hierarchy of classes that inherit from Regexp class the program Pr (see below), when executed will print all the words of the regular language represented by the object o and then it will stop.

```
Pr:  I:=0;
     do
      attach(o);
      for J:=1 to I
      do
       write(WORD(J))
      od;
      writeln;
      if W.B then exit fi
     od
```

**Lemma**

Let i0 be the value of the variable I. Suppose that the some words of the language L(o) were generated by the earlier activations of the **coroutine** o.
An execution of command **attach**(o) has the following effect: the subsequent word of the language L(o) is concatenated to the content of the WORD(1), ... , WORD(I); i.e. the *new* word is placed beginning of the position WORD(I+1). The value of B attribute becomes true iff all the words of the language L(o) were shown.

Proof.
Induction with respect to the length of the expression represented by the object o.

Base. Suppose the actual type of o is ATOM. Then the thesis of the lemma is satisfied.

Induction step. Suppose the lemma holds for every regular expression shorter than an integer k. Consider, a regular expresion of the length k. By our definition it is either a union object or a concatenation object.
case A. Let o be a union object i.e. o **is** UNION. The structure of its commands assures the following

```
        while not exhausted(L)
        do
                attach(L)               -- by induction hypothesis this command returns a word of L
language
        od
        L.B := true                     -- set the exhaustion mark for L
        while not
        do
                attach(R)               -- by induction hypothesis this command returns a word of R
language
        od
        L.R := true
        B := true
```

It is evident that in this way by repeated execution of attach(o) one obtains a sequence of words composed from the all words of L language followed by the sequence of all words from the R language.

case B Suppose the object o is of the class CONCATENATION.

Now the loop of commands of object o assures basically the following

```
      while not exhausted
      do
        store (I);
        attach(L);      -- a word from L
        attach(R);      -- precedes a word from R
        detach;         -- hence a word of (L R) is given
        restore(I)
      od
```

with the necessary reactions to a case when one language (L or R) ends.
It is clear that if the object L and R enjoy the property mentioned in the Lemma then the object o enjoys it too.
This ends the proof of the Lemma. ◆

# Chartres

**program** CHARTRES;

(* An example showing the *generic* class PQ which implements the abstract data type of priority queues *)
(* written by A.Kreczmar 1982 *)
(* Generation of N consecutive prime numbers using the algorithm of E.C. Chartres *)
(* We examine the successive odd numbers. The variable J represents the examined number. Obviously if J is composite then it has a (prime) divisor less than or equal to sqrt(J). Hence, it is enough to test whether J is a multiple of any from T consecutive prime numbers where P(T) {P(T) denotes the T-th prime number} is such that P(T)* P(T) ≤ J . The possible divisors are examined in a tricky way. The priority queue PQ contains the nearest but not less then J multiply of P(K) which is odd. Therefore, if J = min(PQ) then J is a composite number otherwise J is prime *)

```
unit PQ: class(TYPE T; function LESS(X,Y:T):BOOL);
  (* This generic implementation module for Priority Queues with elements of type T uses the notion of heap *)
  (* The least element of PQ is always the first element A[1]        *)
 var A:arrayof T,   (* the table of the heap *)
       N  (* the number of the elements in the heap *)
      ,M  (* the size of the heap *):INTEGER;

 unit DELETEMIN: function:T;
  var I,J:INTEGER,  X:T;
 begin
    if N=0 then return fi;
    result:=A(1);    X:=A(N);      N:=N-1;
    I:=1; J:=2;
    while J <= N
    do  (* update the heap downward *)
       if J+1 <=N then if LESS(A(J+1),A(J)) then J:=J+1 fi fi;
       if LESS(X,A(J)) then exit fi;
       A(I):=A(J);    I:=J;     J:=2*I
    od;
    A(I):=X
 end DELETEMIN;

 unit MIN: function :T;
 begin
    if N =/= 0 then result:=A(1) fi
 end MIN;

 unit INSERT : procedure(X:T);
    var I,J :INTEGER, B:arrayof T;
 begin
   if N=M then (* overflow, increase the space twice *)
     array B dim(1:2*M);      for I:=1 to M do B(I):=A(I) od;
     kill(A);    M:=2*M;      A:=B;
   fi;
```

```
    N, J:=N+1;
    if N=1 then A(1):=X; return; fi;
    I:= J div 2;
    while I>=1
    do  (* update the heap downward *)
       if LESS(A(I),X) then exit fi;
       A(J):=A(I);    J:=I;    I:= J div 2
    od;
    A(J):=X
  end INSERT;

   begin
    M:=2;
    array A dim(1:M);
   end PQ;


begin
  block
     (* we will use ELEM and LESS declared below as actual parameters of the PQ generic class *)

    unit ELEM :class (I,INC:INTEGER);
    end ELEM;

     unit LESS: function(X,Y:ELEM):BOOLEAN;
      (* two ELEM objects are compared with respect to the first attribute *)
     begin
       result:=X.I<=Y.I
     end LESS;

   var X: ELEM;

  begin
   pref PQ(ELEM,LESS) block
     var N, I, T, J, K, ITIME, M: INTEGER;
     var P:arrayof INTEGER; (* an array to store prime numbers*)

     begin
      do
       writeln(" CHARTRES ALGORITHM GENERATING N CONSECUTIVE");
       writeln(" PRIME NUMBERS");
       do
         writeln(" GIVE N:");
         read(N);  writeln(N);
         if N=0 then exit exit else exit fi;
       od;
       ITIME:=TIME;
       array P dim (1:N);
       X:=NEW ELEM(9,6);
       (* 9 IS A MULTIPLY OF 3. THE PURPOSE OF THE SECOND COMPONENT   *)
       (* WILL BE EXPLAINED LATER. X IS THE MINIMUM OF THE PRIORITY QUEUE*)
       (* PQ, BUT IS NOT INSERTED ITSELF INTO IT. CONDITION           *)
```

```
        (* P(T)*P(T) = J WILL BE TESTED OUTSIDE QUEUE ELEMENTS TESTING *)
        P(1):=2;    P(2):=3;    P(3):=5;
        T:=3;    I:=25;    K:=3;    J:=5;
        do
          if N=3 then exit fi;
          J:=J+2;
          if J < X.I  then
            if J=I then  (* J=P(T)*P(T) AND SO ISN'T PRIME *)
              M:= 2*P(T);
              call INSERT( NEW ELEM( I+M , M ));
              (* THE SECOND COMPONENT IS THE INCREMENT FOR THE FIRST ONE   *)
              (* WE INCREASE THE FIRST COMPONENT AND THEN WE INSERT INTO   *)
              (* THE QUEUE THE NEXT ODD MULTIPLY OF P(T)                   *)
              T:=T+1;  I:= P(T)*P(T);
              (* COMPUTE THE NEXT SENTINEL - A  SQUARE OF THE NEXT         *)
              (* PRIME NUMBER.                                             *)
            else  (* J IS PRIME *)
              K:=K+1;    P(K):=J;    if K>=N then exit fi
            fi
          else  (* J=X.I  *)
            do
              (* ALMOST THE SAME AS ABOVE *)
              X.I := X.I + X.INC;
              call INSERT(X);
              X:=DELETEMIN;
              if J=/=X.I then exit fi
              (* WE ARE TO UPDATE ALL ELEMENTS WITH THE SAME COMPONENT X.I *)
            od
          fi
        od;
        ITIME:=TIME-ITIME;
        writeln(" EXECUTION TIME =",ITIME," SEC");
        J:=-1;
        for K:=1 to N
        do
          J:=J+1; if J=10 then writeln; J:=0 fi;
          write(' ',P(K))
        od;
        writeln;
      od;
    end
  end
end
```

# Geo

Program geo;

```
UNIT GEOPLAN : CLASS;
    (* THIS IS A PROBLEM ORIENTED LANGUAGE. IT OFFERS VARIOUS FACILITIES
       FOR PROBLEM SOLVING IN THE FIELD OF ANALTICAL PLANAR GEOMETRY.

       THIS CLASS HAS THE FOLLOWING STRUCTURE:

                    GEOPLAN              <----- CLASS
                   /  |  \
                  /   |   \
                 /    |    \
             POINT    CIRCLE  LINE         <----- CLASSES
            / | \     |     / | \
           /  |  \    |    /  |  \
          /   |   |   |   |   |   \        <--|
         /    |   |   |   |   |    \          |
     EQUALS   DIST  |  | MEETS |    \         |
               ERROR |    |   ERROR     \   OPERA-
                   |     |           /   TIONS
              INTERSECTS  |            |
                         |           |
                       PARALLELTO      <--|


                                          *)
  UNIT POINT : CLASS(X,Y : REAL);
    UNIT EQUALS : FUNCTION (Q : POINT) : BOOLEAN;
    BEGIN
      RESULT:= Q.X=X AND Q.Y=Y ;
    END EQUALS;

    UNIT DIST : FUNCTION (P : POINT) : REAL;
      (* DISTANCE BETWEEN THIS POINT AND POINT P *)
    BEGIN
      IF P = NONE
      THEN
        CALL ERROR
      ELSE
        RESULT:= SQRT((X-P.X)*(X-P.X)+(Y-P.Y)*(Y-P.Y))
      FI
    END DIST;


    UNIT VIRTUAL ERROR : PROCEDURE;
    BEGIN
      WRITELN(" THERE IS NO POINT")
    END ERROR;
```

```
END POINT;


UNIT CIRCLE : CLASS (P : POINT, R : REAL);
  (* THE CIRCLE IS REPRESENTED BY ITS CENTER P AND THE RADIUS R *)

 UNIT INTERSECTS : FUNCTION (C : CIRCLE) : LINE;
  (* IF BOTH CIRCLES INTERSECT AT 2 POINTS, THE LINE JOINING THEM
     IS RETURNED. IF CIRCLES INTERSECT AT ONE POINT, IT IS TANGENT
     TO BOTH OF THEM. OTHERWISE PERPENDICULAR BISECTION
     OF THEIR CENTRES IS RETURNED *)
 VAR R1,R2 : REAL;
 BEGIN
  IF C=/= NONE
  THEN
   R1:= R*R-P.X*P.X-P.Y*P.Y;
   R2:= C.R*C.R-C.P.X*C.P.X-C.P.Y*C.P.Y;
   RESULT := NEW LINE (P.X-C.P.X,P.Y-C.P.Y,(R1-R2)/2);
  FI
 END INTERSECTS;

BEGIN
 IF P=NONE
 THEN
  WRITELN (" WRONG CENTRE")
 FI
END CIRCLE;


UNIT LINE : CLASS (A,B,C : REAL);
 (* LINE IS REPRESENTED BY COEFFICIENTS OF ITS EQUATION AX+BY+C=0 *)


 UNIT MEETS : FUNCTION (L : LINE) : POINT;
  (* IF TWO LINES INTERSECT FUNCTION MEETS RETURNS THE POINT
     OF INTERSECTION, OTHERWISE RETURNS NONE *)
 VAR T  : REAL;
 BEGIN
  IF L =/= NONE AND NOT PARALLELTO (L)
  THEN
   T := 1/(L.A*B-L.B*A);
   RESULT := NEW POINT (-T*(B*L.C-C*L.B), T*(A*L.C-C*L.A));
  ELSE
   CALL ERROR
  FI
 END MEETS;

 UNIT PARALLELTO : FUNCTION (L : LINE) : BOOLEAN;
 BEGIN
  IF L=/= NONE
  THEN
   IF A*L.B-B*L.A = 0.0
```

```
        THEN
          RESULT:=TRUE; WRITELN(" zle");
        ELSE
          RESULT:=FALSE; WRITELN(" dobrze");
        FI
      ELSE
        CALL ERROR
      FI
    END PARALLELTO;

    UNIT VIRTUAL ERROR  : PROCEDURE;
    BEGIN
      WRITELN(" THERE IS NO LINE")
    END ERROR;

  VAR D : REAL;

  BEGIN (* NORMALIZATION OF COEFFICIENTS *)
    D := SQRT(A*A+B*B);
    IF D= 0.0
    THEN
      WRITELN( " ZLE ZERO"); CALL ERROR
    ELSE
      A := A/D;
      B := B/D;
      C := C/D;
    FI
  END LINE;

END GEOPLAN;



BEGIN

  PREF GEOPLAN BLOCK
    (* THE LANGUAGE GEOPLAN IS USED FOR FINDING THE CIRCLE CIRCUMSCRIBED
       ON A GIVEN TRIANGLE:

                          P
                         / \
                        /   \
                       /  .<-\------- CENTRE
                      /       \
                     Q---------R
                                                   *)
  TAKEN POINT,LINE,CIRCLE;

  VAR P,Q,R,CENTRE : POINT,
    L1,L2 : LINE,
    C1,C2,C4 : CIRCLE,
    RADIUS, X1,Y1 : REAL;
```

```
  BEGIN
   DO
    WRITELN(" THIS PROGRAM FINDS THE CIRCUMCENTRE AND THE RADIUS OF ");
    WRITELN(" THE CIRCLE CIRCUMSCRIBED  ON A GIVEN TRIANGLE ");
    WRITELN(" GIVE THE VERTICES COEFFICIENTS OF A TRIANGLE");
    WRITELN(" X1,Y1= ?? ??");
    READ (X1,Y1);
    P := NEW POINT(X1,Y1);
    WRITELN(" ",X1,"   ",Y1);
    WRITELN(" X2,Y2= ?? ??");
    READ (X1,Y1);
    Q := NEW POINT(X1,Y1);
    WRITELN(" ",X1,"   ",Y1);
    WRITELN(" X3,Y3= ?? ??");
    READ (X1,Y1);
    R := NEW POINT (X1,Y1);
    WRITELN(" ",X1,"   ",Y1);

    RADIUS := P.DIST(Q) + Q.DIST(R);

    C1 := NEW CIRCLE (P,RADIUS);
    C2 := NEW CIRCLE (Q,RADIUS);
    C4 := NEW CIRCLE (R,RADIUS);

    L1 := C2.INTERSECTS(C1); (*THE PERPENDICULAR BISECTOR OF THE SIDE PQ*)
    L2 := C2.INTERSECTS(C4); (*THE PERPENDICULAR BISECTOR OF THE SIDE QR *)

    CENTRE := L1.MEETS(L2);

    IF CENTRE = NONE
    THEN
      WRITELN(" ALL POINTS LAY ON THE SAME LINE");
    ELSE
      WRITELN(" THE CIRCUMSCRIBED CIRCLE PARAMETTERS ARE AS FOLOWS:");
      WRITELN(" CENTRE = (",CENTRE.X,',',CENTRE.Y,')');
      WRITELN(" RADIUS = ",CENTRE.DIST(P));
    FI
   OD
  END
END
```

# Heapsort

```
(*-------------------------------------------------------------------*)
(* Heap sort - an "advanced" prefixing example                        *)
(* ---------                                                          *)
(* Reference : A. Szalas, J. Warpechowska : LOGLAN; pages 70-74       *)
(*                                                                    *)
(*  Heap is a particular data structure : it is a kind of tree, which *)
(*  root values are always greater than elements in subtrees.         *)
(*                                                                    *)
(*  The principle of heap sort is simple : at each step we have to find *)
(*  the biggest element and then create the tree with this element on *)
(*  root.                                              *)
(*                                                     *)
(*  The heap is implemented using the array :                   *)
(*    - the root is the first array element                     *)
(*    - left son of element i is stored at 2*i                  *)
(*    - right son of element i is stored at 2*i+1               *)
(*-------------------------------------------------------------------*)
(* Author: Maciek Macowicz (mm@mars.ipl.fr)                     *)
(* Date:   7-march-94                                     *)
(*-------------------------------------------------------------------*)

BLOCK
  UNIT HeapSort: PROCEDURE(INOUT tab: ARRAY_OF Integer);
   VAR Left, Right: Integer;       (* Left and Right sub-table bounds   *)

   UNIT Permute: PROCEDURE(INOUT x,y: Integer);  (* Permute: x,y := y,x *)
    VAR tmp: Integer;
   BEGIN
     tmp:= x;  x:= y;  y:= tmp
   END Permute;

   UNIT PrintTab: PROCEDURE(INPUT msg: STRING); (* print the table 'tab'*)
    VAR                                   (* preceded by 'msg'    *)
     i: Integer;
   BEGIN
     Write(msg, ": ( ");
     FOR i:= Lower(tab) TO Upper(tab) DO  Write(tab(i):4)  OD;
     WriteLn(" )");
   END PrintTab;

   UNIT Sieve: CLASS;
    VAR sieveOver: Boolean,         (* up by derived entities.    *)
        bigElem:   Integer,         (* The biggest (current) element*)
        i, j:      Integer;

    HANDLERS  (* For UNIT Sieve only *)
     WHEN CONERROR:
      WriteLn("CONERROR raised (array index is out of bounds)");
```

```
    WriteLn("UNIT Sieve: CLASS");
    WriteLn("  Left = ", Left);
    WriteLn(" Right = ", Right);
    WriteLn("     i = ", i);
    WriteLn("     j = ", j);
    TERMINATE;
  END HANDLERS;

 BEGIN  (* Sieve *)
   DO
     WriteLn( "----------" );
     INNER;
     CALL PrintTab("      Sieve");
     IF sieveOver                          (* Sieve is over - EXIT    *)
       THEN EXIT FI;
     i:= Left;  (* i points to the biggest element,             *)
     j:= 2*i;         (* j points to the left son of the biggest elem *)
     bigElem:= tab(i);(* contains the biggest value in the table      *)
     WHILE j<= Right DO           (* while inside the table     *)
       IF j< Right AND_IF tab(j)< tab(j+1)
         THEN j:= j+1 FI;
       IF bigElem >= tab(j)
         THEN Exit FI;
       tab(i):= tab(j);
       i:= j; j:= 2*i
     OD;
     tab(i):= bigElem
   OD
 END Sieve;

 UNIT createHeap: Sieve PROCEDURE;
 BEGIN
   CALL PrintTab("Create_Heap");
   sieveOver:= Left= Lower(tab);
   IF NOT sieveOver THEN
     Left:= Left- 1
   FI
 END createHeap;

 UNIT Sort: Sieve PROCEDURE;
 BEGIN
   CALL PrintTab("       Sort");
   sieveOver:= Right= Lower(tab);
   IF NOT sieveOver THEN
     CALL Permute(tab(Lower(tab)), tab(Right));
     Right:= Right- 1
   FI
 END Sort;

BEGIN  (* HeapSort *)
 Left:= Upper(tab) DIV 2 + 1;
 Right:= Upper(tab);
 CALL createHeap;
```

```
    CALL Sort;
  END HeapSort;

 VAR
  t: ARRAY_OF Integer;

 UNIT fillTable: PROCEDURE(INPUT n: Integer; OUTPUT t: ARRAY_OF Integer);
  VAR i: Integer;
 BEGIN
    ARRAY t DIM(1:n);
    FOR i:= 1 TO n DO
     t(i):= ENTIER( 100.0*RANDOM );
    OD
 END fillTable;

BEGIN (* BLOCK *)
  CALL fillTable(8,t);
  CALL HeapSort(t);
END BLOCK;
```

# Gsort

```
(**********************************************************************)
(* File    : gsort.log                                   *)
(* Purpose : Generic sort example                                *)
(* Date    : 20-feb-94                                  *)
(**********************************************************************)

BLOCK

 UNIT Gsort:PROCEDURE(type T; A:ARRAY_OF 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;

 UNIT i_val: CLASS(i: Integer); END i_val;

 UNIT i_lt: FUNCTION (x,y: i_val): Boolean;          (* compare two integers          *)
  BEGIN
   result:= x.i< y.i;
  END i_lt;

 VAR
  tab: ARRAY_OF i_val,                               (* array to sort*)
  i: Integer;                                  (* index                  *)

 BEGIN
  ARRAY tab DIM(1:5);
  tab(1):= new i_val(2);
  tab(2):= new i_val(1);
  tab(3):= new i_val(0);
  tab(4):= new i_val(-1);
  tab(5):= new i_val(-2);

  WriteLn("Before sort: ");
  FOR i:= 1 TO Upper(tab)
  DO
   WriteLn(" ", i, ". ", tab(i).i)
```

```
 OD;

 CALL gsort(i_val, tab, i_lt);

 WriteLn("After sort: ");
 FOR i:= 1 TO Upper(tab)
 DO
  WriteLn(" ", i, ". ", tab(i).i)
 OD;

END;
```

# Pawel

```
program PawelG;   (* author: Paweł Gburzyński, 1983  *)
   var A: arrayof integer;
   var  n, k,   j : integer ;
   unit DrukujA:  procedure;
      var j: integer
   begin
      for  j:=1  to  n  do  write( A(j))  od ;
     writeln
   end  DrukujA;
   unit  F:  procedure ;
      var  i: integer;
   begin
      if  k=n+1  then
           call  DrukujA;
      else
         for  i:= 1  to  n
         do
           if  A[i]=0  then
             A[i] := k; k := k+1;
             call  F;
             k := k-1; A[i]:=0
           fi ;
         od ;
      fi ;
      return
   end  F;
begin
   write(„give n:”);
   readln(n);
   array  A  dim (1:n);
   for  j := 1  to  n  do  A[j] := 0  od ;
   k :=1; ;
   call  F;
   writeln(''Bye``)
end  PawelG
```

# Mat

```
program mat;

unit structures:class;
   signal errorm,errorg,error(i:integer);

   unit ring:class;
      unit virtual add:function(x:ring):ring; end;
      unit virtual mult:function(x:ring):ring; end;
   end ring;

   unit matrix:ring class(n,m:integer);
      var a:arrayof arrayof ring;
      var i:integer;
      unit virtual add:function(x:matrix):matrix;
         var i,j:integer;
         handlers
            when typerror: raise error(3);
         end handlers;
      begin
         if n<> x.n or m <> x.m then raise errorg fi;
         result := new matrix(n,m);
         for i:=1 to n do
            for j := 1 to m do
               result.a(i,j) := a(i,j).add(x.a(i,j))
            od;
         od;
      end add;

      unit virtual mult : function (x:matrix) : matrix;

       var i,j,k : integer ;
       handlers
              when typerror : raise error(3);
       end handlers;

       begin
           if  m<> x.n
              then
                  raise errorm ;
           fi;

           result := new matrix (n,x.m);
           for i := 1 to n do
        for j := 1 to x.m do
        result.a(i,j) := a(i,1).mult(X.a(1,j));
        for k := 2 to m do
         result.a(i,j) :=  result.a(i,j).add( a(i,k).mult( X.a(k,j) )) ;

         OD;OD;OD;

       end mult;

   begin
      array a dim (1:n);
      for i:=1 to n do
         array a(i) dim (1:m);
      od;
   end matrix;
```

```
unit polynomial:ring class(n:integer);
  var a:arrayof ring;

  unit virtual add:function(x:polynomial):polynomial;
     var i,k:integer;
      handlers
         when typerror: raise error(3);
      end handlers;
  begin
     k := imax(n,x.n);
     result := new polynomial(k);
     for i := 0 to imin(n,x.n) do
       result.a(i):= a(i).add(x.a(i));
     od;
     if n>x.n then
       for i:=x.n+1 to n do result.a(i) := a(i) od
     else
       for i := n+1 to x.n do result.a(i) := x.a(i) od
     fi;
  end add;

  unit virtual mult:function(x:polynomial):polynomial;
  var j,i : integer ;
  var b,c : arrayof ring ;
  begin
        array b dim (0:n+x.n) ;
        array c dim (0:n+x.n) ;
       result := new polynomial ( n + x.n );

                       for i:=0 to x.n
                             do
                       c(i):= x.a(i);
                             od;

                       for i:=0 to n
                             do
                       b(i) := a(i) ;
                             od;


       for i:=n+1 to result.n
              do
                 b(i):= a(n+1);
              od;

       for i:=(x.n +1) to result.n
             do
               c(i):= a(n+1);
             od;

       for i:=0 to result.n
             do
                result.a(i) := a(n+1);
             od;

       result.a(0) := b(0).mult(c(0));

       for i:=0 to result.n
             do
        result.a(i):= b(0).mult(c(i));
```

```
          for j:=1 to i
                do
 result.a(i) := result.a(i).add( b(j).mult( c(i-j) ));
                od;
                od;
end mult;


   begin
       array a dim (0:n+1);
   end polynomial;

   unit number:ring class;
      var n:integer;
      unit virtual add:function(x:number):number;
         handlers
            when typerror: raise error(3);
         end handlers;
      begin
         result := new number;
         result.n := n+x.n
      end add;

    unit virtual mult : function (x:number) : number ;

    handlers
           when typerror : raise error(3);;
    end handlers ;

    begin
        result := new number ;
        result.n:= n * x.n;

   end mult;
   end number;


   unit rnumber:ring class;
      var r:real;
      unit virtual add:function(x:rnumber):rnumber;
         handlers
            when typerror: raise error(3);
         end handlers;
      begin
         result := new rnumber;
         result.r := r+x.r
      end add;

    unit virtual mult : function (x:rnumber) : rnumber ;

    handlers
           when typerror : raise error(3);;
    end handlers ;

    begin
        result := new rnumber ;
        result.r:= r * x.r;

   end mult;
   end rnumber;
```

```
    unit complex : ring class ;

    var re,im : real;

    unit virtual add : function (x: complex) : complex ;
     begin
         result := new complex ;
         result.re := re + x.re ;
         result.im := im + x.im  ;
     end add ;

    unit virtual mult : function (x: complex) : complex ;
     begin
         result := new complex ;
         result.re := re*x.re - x.im*im ;
         result.im := re*x.im + x.re*im ;
     end mult ;

    begin
    end complex ;

end structures;

(* ===================================================== *)

begin
  (* system for rings of polynomials and matrices *)

  pref structures block
    var struct:arrayof ring;
    var c:char;

    unit scan:function:char;
    (* reads one character *)
    begin
       result := ' ';
       do
        if result <> ' ' or eoln then exit fi;
        read(result);
       od;
     if eoln and result = ' ' then result:= '&' fi;
    end scan;

    unit num:function(c:char):integer;
    begin
          result := ord(c)- ord('0');
          if result<1 or result >9 then raise error(5) fi;
    end num;

    unit definition:procedure;
      (* definition of the new structure *)
      var i:integer;
    begin
       i := num(c);
       c := scan;
       if c <> ':' then raise error(1) fi;
       struct(i) := gentp;
       call generate(struct(i));
       call inform(struct(i),inf);
    end definition;

    unit information:procedure;
```

```
    var i:integer;
  begin
    c := scan;
    i := num(c);
    if struct(i)=none then raise error(4) fi;
    call inform(struct(i),inf);
  end information;

  unit fill:procedure;
    (* fills  up the structure *)
    var i:integer;
  begin
     c := scan;
     i := num(c);
     if struct(i) = none then raise error(4) fi;
     call inform(struct(i),fillst);
  end fill;


 unit help : procedure;
  (* explains "command language" *)
 var h: char;

  unit riteln : procedure (n : integer);
     var i : integer;
       begin for i := 1 to n
              do
               writeln;
              od;
       end riteln;
begin
     call riteln(4);
      writeln(" HELP");
     call riteln(3);
      writeln (" hit =1= for info about DEFINITION OF STRUCTURE ");
      writeln;
      writeln (" hit =2= for info about OPERATION ON STRUCTURE ");
      writeln;
      writeln (" hit =3= to return to program ");
      writeln;
      writeln ("i n = information about structure no. n");
      writeln;
      writeln ("w n = show structure no. n");
      writeln;
      writeln ("f n = fill structure no. n");
      writeln;
      writeln("$    = exit from program");
      writeln;
      writeln("h    = this text");
    readln;
    read(h);
   case h
      when '1' :call riteln(22);
          writeln("to define structure no. num (1 <= num <= 9) , write :");
                call riteln(2);

                writeln(" num: structype [param1] [param2] <return>");
                writeln("      [substructype] ");
                writeln;
                writeln("where (sub)structype is");
                writeln;
         writeln("  m for matrix 2-dim ; parameters== rows-no.  columns-no.");
```

```
              writeln("                        substructype==elementype.");
              writeln;
              writeln("  p for 1_var_polynomials ; parameter is degree");
       writeln("                            substructype==coefficientype.");
              writeln;
              writeln("  n for integer number ;no parameters");
              writeln("                        no substructure .");
              writeln;
              writeln("  r for real number ;   no parameters");
              writeln("                        no substructure .");
              writeln;
              writeln("  c for complex number ;no parameters");
              writeln("                        no substructure .");

     when '2' : call riteln(22);

writeln("Operations work on two structures ( of the same type only )");
     writeln;
     writeln("producing a third structure .");
     writeln;
     writeln("Command is : ");
     writeln("               o res=op1 @ op2 ");
     call riteln(3);
writeln("where res,op1,op2 are structure-identifiers <numbers>");
     writeln;
     writeln("and @ is the operation sign ( + , * ) .");
     call riteln(3);
     writeln(" Examples :");
     writeln("           o 3=1*2");
     writeln("(struct-1 * struct-2 --> struct-3)");
     writeln;
     writeln("           o 2=1+2");
     writeln("(struct-1 + struct-2 --> struct-2)");
     otherwise  writeln("                 bye !");

  esac;
end help;


 unit writestruct:procedure;
  (* displays the structure *)
    var i:integer;
 begin
   c := scan;
   i := num(c);
   if struct(i) = none then raise error(4) fi;
   call inform(struct(i),wrt);
 end writestruct;

 unit operation:procedure;
 (* performs an operatin *)
   var i,k:integer;
 begin
   c := scan;
   k := num(c); (* result *)
   c := scan;
   if c <> '=' then raise error(2) fi;
   c := scan;
   i := num(c); (* first argument *)
   c := scan; (* operation *)
   case c
when '+': call  opadd(k,i);
```

```
 when '*': call  opmult(k,i);
 otherwise  raise error(5);
    esac;
  end operation;

  unit gentp:function:ring;
   (* generates the pattern for the new structure *)
    var n,m:integer;
  begin
     c := scan;
       case c
   when 'm': (* matrix *)
             read(n); read(m); readln;
             if n<1 or m<1 then raise error(7) fi;
             result := new matrix(n,m);
             result qua matrix.a(n,m) := gentp;
   when 'p': (* polynomial *)
             read(n); readln;
             if n<0 then raise error(7) fi;
             result := new polynomial(n);
             result qua polynomial.a(0) := gentp;
    when 'n': (* number *)
             result := new number;
    when 'r': (* rnumber *)
             result := new rnumber;
    when 'c': (* complex *)
             result := new complex;
   otherwise raise error(6);
       esac;
  end gentp;

  unit generate:procedure(x:ring);
   (* generates the structure *)
    var y:ring,
        i,j:integer;
  begin
     if x is matrix then
         y := x qua matrix.a(x qua matrix.n ,x qua matrix.m);
         for i:=1 to x qua matrix.n do
           for j:=1 to x qua matrix.m do
               x qua matrix.a(i,j) := copy(y);
               call coppy(x qua matrix.a(i,J),y);
               call generate(x qua matrix.a(i,j));
           od;
         od
      else
         if x is polynomial then
            y := x qua polynomial.a(0);
            for i:=0 to (x qua polynomial.n)+1 do
                x qua polynomial.a(i) := copy(y);
                call coppy(x qua polynomial.a(i),y);
                call generate(x qua polynomial.a(i));
            od;
         fi
      fi
  end generate;

 unit coppy:procedure(x,y:ring );
  var i:integer;
begin
   if x is matrix then
      x qua matrix.a := copy(y qua matrix.a);
```

```
      for i := 1 to x qua matrix.n do
         x qua matrix.a(i) := copy (y qua matrix.a(i));
      od
    else
    if x is polynomial then
       x qua polynomial.a := copy(y qua polynomial.a);
     fi
   fi
end;

  unit inform:procedure(x:ring; procedure op(x:ring));
    (* auxilliary *)
  begin
       if x = none then raise error(4) fi;
       if x is matrix then
         write("  matrix ", x qua matrix.n:3, " X ",x qua matrix.m:3);
         writeln;
         call  op(x)
       else
         if x is polynomial then
            write("  polynomial deg ", x qua polynomial.n:3);
            writeln;
            call op(x)
         else
            if x is number then write("  number ");
                                call op(x);
                           else
            if x is rnumber then write("  realnumber ");
                                 call op(x);
                            else
            if x is complex then write(" complex number ");
                                 call op(x);
            fi;
            fi;
            fi;
           fi
       fi;
     writeln
  end inform;

  unit fillst:procedure(x:ring);
    (* fills the  structure *)
     var i,j:integer,
           y:ring;
  begin
     if x is matrix then
        for i:=1 to x qua matrix.n do
          writeln(" row ",i:3);
          for j := 1 to x qua matrix.m do
             writeln(" column",j:3);
              call inform(x qua matrix.a(i,j),fillst);
          od;
        od
     else
        if x is polynomial then
           for i := 0 to x qua polynomial.n do
              writeln (" element ",i:3);
                call inform(x qua polynomial.a(i),fillst);
           od
        else
           if x is number then read(x qua number.n);
            else
```

```
                if x is rnumber then read (x qua rnumber.r);
                 else if x is complex then
                                     write ( "real ");
                                     read ( x qua complex.re );
                                     write ( "imag. ");
                                     read ( x qua complex.im );
                      fi;
                fi;
             fi;
          fi;
       fi;
    end fillst;

    unit inf:procedure(x:ring);
      var y:ring;
    begin
      if x is matrix then y := x qua matrix.a(1,1); call inform(y,inf)
      else
        if x is polynomial then y := x qua polynomial.a(0); call inform(y,inf)
fi
      fi;
      writeln;
    end inf;

    unit wrt:procedure(x:ring);
     (* displays the structure *)
       var i,j:integer,
             y:ring;
    begin
       if x is matrix then
          writeln;
          for i := 1 to x qua matrix.n do
            for j := 1 to x qua matrix.m do
            write(" row ",i:3," ,column ",j:3,":");writeln;
               y := x qua matrix.a(i,j);
               call inform(y,wrt);
               writeln;
            od
          od
        else
           if x is polynomial then
              writeln;
              for i := 0 to x qua polynomial.n do
                write(" element ",i:3,':');
                y := x qua polynomial.a(i);
                call inform(y,wrt);
                writeln;
               od
           else
              if x is number then
              write(x qua number.n:4);
              else
                 if x is rnumber then
                 write(x qua rnumber.r);
                 else
                    if x is complex then
                 write("( ",x qua complex.re," ) + i*( ",x qua complex.im," )");
                    fi;
                 fi;
              fi;
           fi
        fi
```

```
   end wrt;

   unit opadd:procedure(k,i:integer);
   (* auxilliary - performs addition *)
      var x,y:ring,
          j:integer;
      handlers
         when typerror: raise error(3);
      end handlers;
   begin
      c := scan;
      j := num(c);
      struct(k) := struct(i).add(struct(j))
    end opadd;

   unit opmult:procedure(k,i:integer);
   (* auxilliary - performs multiplication *)
      var x,y:ring,
          j:integer;
      handlers
         when typerror: raise error(3);
      end handlers;
   begin
      c := scan;
      j := num(c);
      struct(k) := struct(i).mult(struct(j))
    end opmult;

    handlers
      when error: write("  **** error **** ");
                   case i
           when 0: writeln(" undefined statement ");
           when 1: writeln(" ':' expected");
           when 2: writeln(" '=' expected ");
           when 3: writeln(" not compatible types in operation");
           when 4: writeln(" structure not defined ");
           when 5: writeln(" illegal operation");
           when 6: writeln(" unrecognized structure");
           when 7: writeln(" wrong parameter");
                   esac;
                   wind;
      when errorm:
                  writeln(" matrix dimensions - in mult - are not compatible");
                           wind;
      when errorg:
                  writeln(" matrix dimensions - in sum - are not compatible");
                  wind;
    end handlers;

 begin  (* main program *)
   array struct dim(1:9);
   do
     write(" ** ");
     c := scan;
     case c
   when '1','2','3','4','5','6','7','8','9': call definition;
   when 'f': call fill;
   when 'w': call writestruct;
   when 'o': call operation;
   when '$': exit;
   when 'i': call information;
   otherwise call help;
```

```
      esac;
      readln; writeln;
   od;writeln;writeln("                            bye .");
 end
end
```

# Winograd

```
program winograd;

signal Niezgoda;

unit Winograd: procedure(A,B: array_of array_of real;
                                        output C: array_of array_of real);
(* require  macierze A i B są kwadratowe rozmiaru n x n  *)
(* ensure  macietz C jest produktem macierzy A i B *)
   var i,j,k,n,m: integer,
       W,V: array_of real,
       p: boolean,
       s: real;

begin
(* ustalic czy macierze moga byc mnozone tzn.
   czy ilosc wierszy w A = ilosc kolumn w B? *)
(* ustalic czy n jest parzyste? *)
(* obliczyc "preprocessing" *)



if lower(A) <> lower(B) or upper(A)<>upper(B) then raise Niezgoda fi;
i := upper(A);
j := lower(A);
k := i-j;
for l : = j to i do
      if lower(A(l)) <> lower(B(l)) or upper(A(l))<>upper(B(l)) then raise Niezgoda fi;
od;

(* mozna mnozyc *)

n := k+1;
p := (n mod 2) = 0; (* p ≡ n jest parzyste?  *)
m := n div 2;
array W dim (1:n);
array V dim (1:n);
array C dim (1:n);
for i := 1 to n
do
  array C(i) dim (1:n)
od;
(* obliczanie "preprocessingu" *)
for j:= 1 to n
do
  s:=0;
  for i := 1 to m
  do
    s := A[j,2*i-1] * A[j,2*i] +s;
```

```
  od;
  W[j] := s;
od;
(* Lemat 1
Dla każdego j, 1≤ j ≤ n,   W_j = Σ_{i=1}^{n÷2} A_{j,2i-1} * A_{j,2i}
*)
for j:= 1 to n
do
  s:=0;
  for i := 1 to m
  do
    s := B[2*i-1,j) * B(2*i,j] +s;
  od;
  V[j] := s;
od;
(* Lemat 2
Dla każdego j, 1≤ j ≤ n,
  V_j = Σ_{i=1}^{n÷2} B_{2i-1,j} * B_{2i,j}
                                          *)
(* obliczanie iloczynu macierzy *)
for i := 1 to n    (* dla każdego  1≤ i ≤ n   *)
do
  for j := 1 to n    (* dla każdego  1≤ j ≤ n  *)
  do
    s:= 0;
    for k:= 1 to m
    do
      s:= (A[i,2*k-1]+B[2*k,j]) * (B[2*k-1,j]+A[i,2*k]) +s;
    od;
    (*Lemat 3 Dla każdych wartości i,j, 1≤ i ≤ n, 1≤ j ≤ n,
   s = Σ_{k=0}^{n÷2} (A_{i,2k-1} + B_{2k,j}) * (B_{2k-1,j} + A_{i,2k})
   *)
    C[i,j] :=s-W[i]-V[j];
(* Lemat 4   Dla każdych wartości i,j, 1≤ i ≤ n, 1≤ j ≤ n,
   C_{i,j} = Σ_{k=0}^{2*(n÷2)} A_{i,k} * B_{k,j}
*)
    if not p    (*   poprawiamy - gdy n jest nieparzyste  *)
    then
       C[i,j] := C[i,j] + A[i,n]*B[n,j];    C_{i,j} = (Σ_{k=0}^{2*(n÷2)} A_{i,k} * B_{k,j}) + A_{i,n} * B_{n,j}
    fi;
   C_{i,j} = Σ_{k=0}^{n} A_{i,k} * B_{k,j}
  od; (* j *)
od;   (* i *)
end Winograd;

var M1,K1,J1,J2 : array_of array_of real,
    i,j,n,t,k : integer,
```

Lemat 1: $W_j=\sum_{i=1}^{n\div 2} A_{j,2i-1}*A_{j,2i}$

Lemat 2: $V_j=\sum_{i=1}^{n\div 2} B_{2i-1,j}*B_{2i,j}$

Lemat 3: $s=\sum_{k=0}^{n\div 2}(A_{i,2k-1}+B_{2k,j})*(B_{2k-1,j}+A_{i,2k})$

Lemat 4: $C_{i,j}=\sum_{k=0}^{2*(n\div 2)} A_{i,k}*B_{k,j}$

$C_{i,j}=(\sum_{k=0}^{2*(n\div 2)} A_{i,k}*B_{k,j})+A_{i,n}*B_{n,j}$

$C_{i,j}=\sum_{k=0}^{n} A_{i,k}*B_{k,j}$

```
    s: real;


(* ------------------------------------------------------------------------ *)
begin (*programu *)
   write("podaj wartosc n=");
   readln(n);writeln(n);
   array M1 dim (1:n);
   array K1 dim (1:n);
   array J1 dim (1:n);
   array J2 dim (1:n);
   for i := 1 to n do
     array M1[i] dim (1:n);
     array K1[i] dim (1:n);
     array J1[i] dim (1:n);
     array J2[i] dim (1:n);
   od;
   for i := 1 to n do
   for j := 1 to n do
     if i=j then M1[i,j]:= 1 else M1[i,j]:=0 fi;
     K1[i,j]:= (i-1)*n+j;
   od;
   od;

   t:=time;
   call Winograd(M1,K1,J1);
   t:=time - t;
   writeln("czas Winograd=",t:6);

   t:=time;
   for i:=1 to n do
   for j:=1 to n do
     s:=0;
     for k:=1 to n do
       s := s + M1[i,k]*K1[k,j];
     od;
     J2[i,j]:=s;
   od;
   od;
   t:=time-t;
   writeln("czas normalnego mnozenia=",t:6);

   for i := 1 to n do
     writeln;
     for j := 1 to n do
       write(J1[i,j]:5:0);
     od;
   od;



end program;
```

# Towhanc

```
block
(* towers of hanoi *)

(* there are three towers built of decreasing rings stringed onto sticks    *)
(* at the initial state all rings are stringed onto stick no. 1. our job is *)
(* to move all rings from the stick 1 to the stick 3. the difficulty is     *)
(* that we mustn't violate the following conditions                         *)
(*    1. we can move only one ring at one step                              *)
(*    2. each ring may be placed only onto a greater one                    *)
(* to manage with this difficult problem we have an auxilliary stick 2      *)

 unit wz:coroutine(n,f,t:integer);
   (* move n rings from stick f to stick t *)
      var k:integer;
  begin
   return;
   do
   k:=6-(f+t);
   if n>1 then  attach (p(n-1,f,k)); fi;
   call modyf(f,t);  (* move only one ring *)
   if n>1 then attach (p(n-1,k,t)); fi;
    detach;
    od;
   end wz;

unit modyf:procedure(f,t:integer);
   (* move the topmost ring from stick f to stick t *)
 begin
   top(t):=top(t)+1;
   w(t,top(t)):=w(f,top(f));
   w(f,top(f)):=0;
   top(f):=top(f)-1;
   call displ;
 end modyf;

unit displ:procedure;
  (* printing *)
 var t,i,j,k,m,n:integer;
  begin
   t:=1;
   for i:=2 to 3 do
    if top(i)>top(t) then t:=i fi od;
   t:=top(t);
   for i:=t downto 1 do
    m:=15;
    for j:=1 to 3 do
     for k:=1 to m do write(" "); od;
     if w(j,i)=/=0 then for k:=1 to w(j,i) do write("*") od;
```

```
     fi;
     m:=15-w(j,i);
    od;
    writeln;
  od;
  for i:=1 to 15 do write(" "); od;
  for i:=1 to 45 do write("-"); od;
  writeln;
 end displ;

 var  w:arrayof arrayof integer,   (* how many rings are stringed *)
                                   (* on each stick               *)
      top:arrayof integer,  (* the topmost ring size on each stick *)
      nb,i,j,k,timeb:integer,
      p:arrayof arrayof arrayof wz; (* coroutine pointers *)

   begin
    array w dim(1:3);
    array top dim(1:3);
    writeln(" program towers of hanoi");
    writeln(" version with coroutines");
    do writeln(" give the number of rings");
       read(nb);
       writeln(nb);
       if nb>0 then exit else writeln(" number of rings must be greater than 0")
    fi od;
    timeb:=time;
    top(1):=nb;
    array w(1) dim(1:nb);
    array w(2) dim(1:nb);
    array w(3) dim(1:nb);
    k:=nb;
    for i:=1 to nb do w(1,i):=k;
                      k:=k-1;
    od;
    (* stick 1 is full *)
    writeln(" the algorithm acts as follows");
    call displ;
    array p dim (1:nb);
    for i:=1 to nb
    do array p(i) dim(1:3);
      for j:=1 to 3
      do array p(i,j) dim(1:3);
         for k:=1 to 3
         do if j=/=k then p(i,j,k):=new wz(i,j,k) fi
         od
      od
    od;
    attach (p(nb,1,3));
    writeln(" execution time for",nb:4," rings =",time-timeb," sec");
end
```

# Differ

```
block  (* SYMBOLIC DIFFERENTIATION *)
 (* Overloading? *)
(* THIS IS AN EXAMPLE OF STEPWISE REFINEMENT PROGRAMMING INTENSIVELY USING  *)
(* VIRTUALS. WE COMPUTE THE DERIVATIVE OF AN ALGEBRAIC EXPRESSION. THE EXP- *)
(* RESSION IS REPRESENTED IN THE FORM OF A TREE. LITERALS REPRESENTING *)
(* CONSTANTS AND VARIABLES ARE LOCATED IN THE LEAVES. THE OTHER NODES ARE   *)
(* OPERATORS. while IMPLEMENTING DIFFERENTIATION OF ANY PARTICULAR KIND OF  *)
(* EXPRESSION WE NEED NOT INTEREST IN THE OTHER EXPRESSIONS KINDS. DUE to *)
(* THE ADVENTAGES OF VIRTUALS WE ONLY NEED EXPRESSION to HAVE A function *)
(* NAMED "DERIV" WHICH RETURN ITS DERIVATIVE. THE EXACT FORM OF THIS *)
(* function IS COMPLETELY OUT OF OUR INTEREST AND MAY BE DEFINED SEPARATELY *)
(* AND INDEPENDENTLY for EACH KIND OF EXPRESSION                              *)


  unit RSYMBOL:class;
    (* DIFFERENTIATION OF A function OF A VARIABLE X *)

      unit EXPR:class; (* OUR FUNCTIONS WILL BE EXPRESSIONS *)
          unit virtual DERIV:function(X:VARIABLE):EXPR;
                        end DERIV;
              end EXPR;

      unit CONSTANT:EXPR class(K:REAL);
         (* DIFFERENTIATED EXPRESSION WILL CONSIST OF CONSTANT *)
          unit virtual DERIV:function(X:VARIABLE):EXPR;
                        begin
                          result:=ZERO;
                        end DERIV;
                        end CONSTANT;

      unit VARIABLE:EXPR class(ID:STRING);
         (* DIFFERENTIATED EXPRESSION WILL OBVIOUSLY CONSIST OF VARIABLES*)
          unit virtual DERIV:function(X:VARIABLE):EXPR;
                        begin
                          if X=THIS VARIABLE then
                            result:=ONE
                          else
                            result:=ZERO
```

```
                  (*THIS IS THE DERIVATIVE OF A VARIABLE
                   OTHER then X WITH RESPECT to X       *)
                 fi
               end DERIV;
        end VARIABLE;

unit PAIR:EXPR class(L,R:EXPR);
   (* WE WILL ALSO COMPUTE DERIVATIVES OF EXPRESSIONS WITH TWO
     ARGUMENT OPERATORS                                   *)
   unit virtual DERIV: function(X:VARIABLE):EXPR;
   end;
        end PAIR;


unit SUM:PAIR class;
   (* WE DIFFERENTIATE THE SUM OF TWO EXPRESSIONS *)
    unit virtual DERIV:function(X:VARIABLE):EXPR;
               var LPRIM,RPRIM:EXPR;
               begin
                 LPRIM:=L.DERIV(X);
                 RPRIM:=R.DERIV(X);
                 (*WE DELETE 0 AS THE NEUTRAL ELEMENT OF
                  ADDITION                         *)
                 if LPRIM=ZERO then
                   result:=RPRIM
                 else
                 if RPRIM=ZERO then
                   result:=LPRIM
                 else
                   result:=NEW SUM(LPRIM,RPRIM)
                 fi
                 fi;
               end DERIV;
       end SUM;


 unit DIFF:PAIR class;
    (* WE DIFFERENTIATE THE DIFFERECE OF TWO EXPRESSIONS *)
     unit virtual DERIV:function(X:VARIABLE):EXPR;
                var LPRIM,RPRIM: EXPR;
                begin
                  LPRIM:=L.DERIV(X);
                  RPRIM:=R.DERIV(X);
                  (* WE DELETE THE SUBTRACTED ZERO *)
                  if RPRIM=ZERO then
                    result:=LPRIM
                  else
                    result:=NEW DIFF(LPRIM,RPRIM)
                  fi
                end DERIV;
        end DIFF;
```

```
 unit DISPLAY:procedure(T:STRING,E:EXPR);
   (* DISPLAY THE EXPRESSION TREE IN A READABLE FORM *)

        unit SCAN:procedure(E:EXPR);
            (* Compare the old style procedure scan with the virtual method DERIV.
               Remark, that once you add a new kind of expression you must rewrite SCAN *)
              begin
               if E IS SUM then
                 WRITE(" ("); CALL SCAN(E QUA PAIR.L);WRITE("+");
                 CALL SCAN(E QUA PAIR.R);
                 WRITE(" )")
               else
               if E IS DIFF then
                  WRITE(" (");
                 CALL SCAN(E QUA PAIR.L);WRITE("-");
                 CALL SCAN(E QUA PAIR.R);
                  WRITE(" )")
               else
               if E IS CONSTANT then
                 WRITE(E QUA CONSTANT.K:6:2)
               else
               if E IS VARIABLE then
                 WRITE(E QUA VARIABLE.ID);
                fi fi fi fi;
          end SCAN;

            begin
               WRITE(T);
               CALL SCAN(E); WRITELN;
  end DISPLAY;


 var ZERO,ONE:CONSTANT;
 begin (* INITIALIZE LITERAL PATTERNS OF CONSTANTS *)
  ZERO:=NEW CONSTANT(0);
  ONE:=NEW CONSTANT(1);
 end RSYMBOL;


begin
 pref RSYMBOL block
      var X,Y,Z:VARIABLE,U,V,E,F:EXPR;
      begin
        X:=NEW VARIABLE("X");Y:=NEW VARIABLE("Y");
        U:=NEW SUM(X,Y);Z:=NEW VARIABLE("Z");
        V:=NEW DIFF(Z,NEW CONSTANT(4));
        E:=NEW DIFF(U,V); E:=NEW SUM(E,NEW DIFF(X,Y));
        CALL DISPLAY(" EXPRESSION E= ",E);
        F:=E.DERIV(X);
        CALL DISPLAY(" DERIVATIVE WITH RESPECT to X:",F);
      end
end
```