APL and Algol68, the correspondence and the differences, especially in applications of graph-analysis

P.L.J. Siero

Leiden University Computer Centre
The Netherlands

Abstract.

In this presentation the characteristics of both languages will be presented. Typical features and facilities of Algol68 will be compared with the most important facilities of APL. Some other complicated features of Algol68 are considered and their possibilities for applications.
Applications in the field of graph theory have been coded for the two languages.

Introduction.

In this lecture a number of characteristics of APL and Algol68 will be treated.
Main features of both languages will be considered; afterwards the characteristics of both languages are compared.
Much attention will be given to the types of datastructures in APL and the complicated datastructures of Algol68.
APL operators will be compared with Algol68 operators, the possibility of new operator definitions in Algol68 will be described.
APL functions will be considered and attention will be drawn to Algol68 procedures with their many possibilities.
After these considerations about the features of the languages, applications are shown in the area of graph theory and numerical analysis.

Par. 1.

The APL language, datatypes, operators, functions.

APL uses rectangular datatypes for all dimensions, scalars, vectors, matrices and so on. The contents of the data can be of the type numerical or of the type character.
It is not possible to work with complex numbers. Up till now there are no implementations of nested arrays or mixed arrays.
Working with lists is rather difficult. It is possible to consider rows of a certain matrix as elements of lists. The last position in the row is used as a pointer to the next element of the list (see fig. 1). It is possible to write APL functions to delete elements of the list, to add elements, to find the last element of the list.
Of course there should be an agreement about the contents of the pointer location of the last element. It is also possible to refer from one matrix to another.

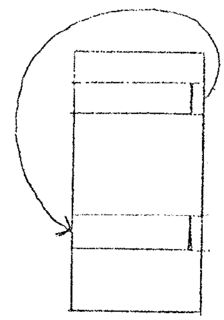


fig. 1

The operators in APL make use of the normal, scalar dyadic and monadic functions +, -, x ..., the result is again a function.
For instance:

```
a.    A ← 4  3  7
   +\A gives 4   7  14
   x\A gives 4  12  84

   (\ is the scan operator)

b. M ← 2 3 ρ 1 2 1 2 1 2
   M+.xA gives  17  25

   (. is the inner product operator)
```

There are three more operators, namely: reduction, axis and outer product.
There are a great number of primitive functions in APL (look at the keyboard of the terminals) and moreover there are so called defined functions. The primitive functions, as the defined functions, are niladic, monadic or dyadic. They are working on the APL data structures and the result is also a data structure. Sometimes the APL functions do not have an explicit result (niladic).
At the end of this review the possibilities of jumps and labels have to be mentioned.

Par. 2.

The language Algol68, datatypes, operators, functions.

Before talking about these features it should be mentioned that APL is an interpreter, and a timesharing tool with many possibilities of conversational use. For instance function editing can easy be done from at the terminal. Algol68, however, is a batch application and it has a much wider range of possibilities.

The notation of the ALGOL68C compiler of Cambridge is used, basic symbols are written with one accent in front of them.

2.1.

In Algol68 there are many possibilities of data structures, look at the following declarations:
First one has rectangular datatypes, for instance: 'int n;

'loc (1:5) 'int m;

example m:=(2,6,3,4,1);
in Algol68 nomenclature this is called 'row 'of 'int
'loc stands for local declaration.

'loc (1:3) 'char a;

example a:=("p","q","r");
this is called 'row 'of 'char

'loc (4:10,3:8) 'real x;

this is called 'row 'row 'of

There are other possibilities to use non-rectangular datatypes, for instance ((1,2,3),(4,5),(6,7,8,9)), in Algol68 this is called 'row 'of 'row
More examples can be made by using structures for instance putting an integer and a real together in
'struct('int a,'real b) st; see figure 2.

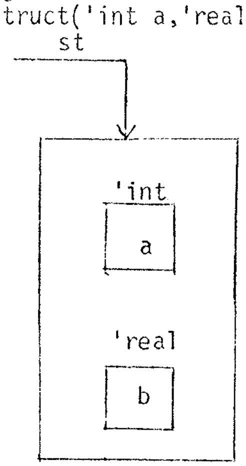


fig. 2

Or if one wants to administer names, ages and weights.

'struct ((30) 'char name,'int age,weight)u;

Looking at the example of the integer and the real, you sometimes want to combine more integers with more reals, for instance ((3,2.1),(4,3.2),(6,8.7)), this is called a 'row 'of 'structures in Algol68.

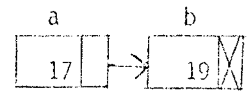
Another important Algol68 notion is the term mode; there are standard modes, as int, real, bool.
The programmer can make his own modes, as 'mode 'new='struct('int a,'real b); and 'mode 'node='struct((2) 'int name, 'ref 'node next);

This is circular mode declaration, 'node appears to the right and to the left of the = token.
The definition of mode 'node can be used in list processing.

For instance:

```
'node a:=(17,nil);
'node b:=(19,nil);
 next 'of a:=b;
     gives
```

fig. 3



In Algol68 there is a principle of orthogonality of modes and programmers can construct new modes out of already existing modes.

2.2.

The normal operators +,-,... between scalars can be expanded, to work with more complicated data structures.
An example can be given by the operator + working between two rows, having as a result a new row:

```
'OP + = (( )'REAL A,B)( )'REAL:
'BEGIN
'LOC (1: 'UPB A)'REAL SUMR;
'FOR I 'TO 'UPB A 'DO
SUMR(I):=A(I)+B(I)
   'OD;
SUMR 'END;
```

Some remarks about the coding:
The operator + has two parameters of the type 'row 'of 'real and the result is of the same type, this is a so-called dyadic operator.

'row 'of can be coded as ()
In the definition of the operator + a local declaration of a 'row 'of 'real is present, namely the definition of sumr. There is also a loop clause. No further declaration of the variable i is needed, because i appears just after FOR; this is an Algol68 rule.
'upb is a standard Algol68 operator, which gives the upperbound of an array, there is also an operator 'lwb for the lowerbound of an array.
If the use of the scan operator of APL has to be demonstrated in Algol68, you can do so as follows for the case +\ (applied to a 3 by 3 matrix):

```
'OP 'SCAN=((,)'REAL A)(,) 'REAL:
'BEGIN
'LOC (1:3,1:3) 'REAL SUMM;
SUMM( ,1):=A( ,1);
SUMM( ,2):=A( ,1)+A( ,2);
SUMM( ,3):=SUMM( ,2)+A( ,3);
  SUMM 'END;
```

Remarks: there is one parameter in the operator scan, the parameter and the result have both the type 'row 'row.

A(,1) means the first column of A.
The + operator in the operator SCAN is the just defined operator +. An other example can be made for the case x\, using a suitable definition of an operator x.
Also the second example of par. 2, the multiplication of a matrix with a vector can be done in Algol68.
See reference 1.

2.3.

Procedures in Algol68 give many possibilities; in practice, there are standard procedures and user written procedures. The procedures are working with datastructures and their results are datastructures as well. One can call this the traditional use. There are many other possibilities to manipulate with procedures, in the same way as one can work with integers, characters, etc.
One can construct rows of procedures, procedures with other procedures as parameters.
There is much resemblance between the working of operators and procedures in Algol68. A dyadic (monadic) operator can be compared with a procedure with two (one) parameter(s). In the case of operators however, one often has to take care of priorities if the same operator is twice defined for different datastructures.

Some examples of procedures:

(a)

The procedure MEMB decides if an integer Q belongs to an array P or not.

```
'PROC MEMB=(( ) 'INT P,'INT Q) 'BOOL:
'BEGIN 'BOOL FOUND:= 'FALSE;
'FOR 1 'FROM 'LWB P  'TO 'UPB P 'WHILE  'NOT
FOUND
'DO (Q=P(1)/FOUND:='TRUE ) 'OD; FOUND 'END;
```

Remarks: the parameters are a row of integers and an integer, the result is of the type boolean.
In this example there is again a so-called loop clause:

(Q=P(I)/FOUND:='TRUE) is an abbreviation of
'IF Q=P(I) 'THEN FOUND:=TRUE 'FI

This is a so-called conditional clause. Loop clauses and conditional clauses are examples of the working of the control structures in Algol68.
There is much more to say about it; it has to do with "ranges and reaches" and the scope and availability of identifiers.

I finish this chapter with another example of an Algol68 'procedure COMPR which uses the procedure MEMB.
The procedure COMPR has two parameters, an integer array and an integer. The input array P has been filled with non zeros in the beginning and zeros at the end. The result is an array identical to the input array without duplicates of non zeros.

(b)

```
 'PROC COMPR=(( ) 'INT P,'REF 'INT J)( ) 'INT:

'BEGIN
  'LOC (1: 'UPB P) 'INT H;
  J:=1; H(1):=P(1);
 'FOR I 'FROM 2 'TO 'UPB P 'DO
  'IF MEMB(H,P(I))='FALSE 'THEN J:=J+1;
H(J):=P(I) 'FI
 'OD;
 H
'END;
```

Remark:
The number of elements $\neq 0$ in the result is delivered by the parameter j, in the procedure heading this has been coded as 'REF 'INT J, which means to be a pointer to an integer value.

Par. 3.

Applications of APL in the area of graphs.

There are some programming languages which have been developed to be suitable for graph manipulations.
But also APL seems to be a nice language for these purposes.
As an example I mention the following application.
In a connected graph G with at least one edge one can find a closed Eulerian line if the degree of each of its vertices is even. A closed Eulerian line - or Euler path - is a closed line which contains all the edges of G (all lines are walked through one and only one time).

Example:

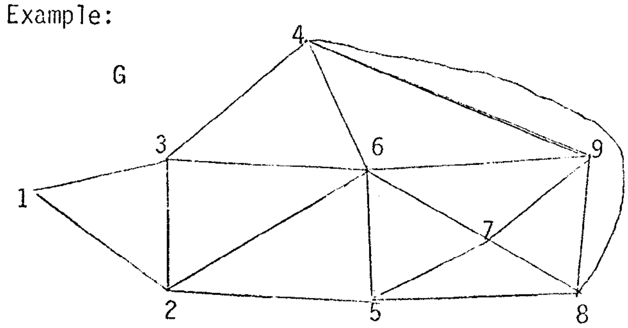


fig. 4

If we start at vertex 2, we can find the following Euler path:

2 6 3 4 9 6 7 5 8 7 9 8 4 6 5 2 3 1 2,

using 3 APL functions namely COMP, EULER and PATH.

Explanation.

The graph G can be represented by the following adjacency matrix ZA:

```
0 1 1 0 0 0 0 0 0
1 0 1 0 1 1 0 0 0
1 1 0 1 0 1 0 0 0
0 0 1 0 0 1 0 1 1
0 1 0 0 0 1 1 1 0
0 1 1 1 1 0 1 0 1
0 0 0 0 1 1 0 1 1
0 0 0 1 1 0 1 0 1
0 0 0 1 0 1 1 1 0
```

By means of the function COMP one can decide if the graph is connected or not (in our case the answer is yes). ZA has to be converted to a so called adjacency list. In our case this list consists of:

2 3
1 3 5 6   and so on.

In the function COMP this information has been stored in the matrix M in a special way. With a well known algorithm you can see if the number of compounds is 1 or not. If so and and the sum of every row (or column) is even, the function Euler and Path are necessary for the calculation of an Euler path, starting with a certain vertex.
The functions Euler and Path (the last one is recursive) have been written in APPENDIX A.
The function COMP, which checks for connectivity of the graph, follows:

```
        ∇COMP[⎕]∇
        ∇ COMP ZA;N;V;M;T;C;FL
[1]   ⍝ IS THIS GRAPH CONNECTED?
[2]   ⍝VERTICES ARE COLLECTED
[3]   ⍝  IN V  AND C
[4]    N←1
[5]    M←((1↑⍴ZA),10)⍴0
[6]   LOOP:FL←ZA[;N]/⍳1↑⍴ZA
[7]    M[N;⍳+/(FL>N)]←(FL>N)/FL
[8]    →((N←N+1)≤1↑⍴ZA)/LOOP
[9]    FL←0
[10]   T←N←1
[11]   C←1↓T
[12]   V←,1
[13]  ⍝TRY TO ADD TO VECTOR V
[14]  P1:→((V[N]∊C)=1)/LL
[15]   C←C,V[N]
[16]   V←V,((M[V[N];]≠0)/M[V[N];])
[17]  LL:→(((N←N+1)≤T),T=⍴V)/P1,P2
[18]  ⍝ IF SOMETHING ADDED,ADD MORE
[19]   T←⍴V
[20]   →P1
[21]  P2:N←1
[22]  B:→(((N∊C)=1),(+/(M[N;]∊V))>0)/R,A
[23]  R:→(((N←N+1)≤1↑⍴ZA),FL=0)/B,CO
[24]  ⍝IF FL=1 GO TO P2 TO GET MORE
[25]   FL←0
[26]   →P2
[27]  A:FL←1
[28]   V←V,((M[N;]≠0)/M[N;]),N
[29]   C←C,((M[N;]≠0)/M[N;]),N
[30]   →R
[31]  CO:V←((V⍳V)=⍳⍴V)/V
[32]  ⍝ COMPRESSION OF VECTOR
[33]   →((⍴V)<1↑⍴ZA)/L
[34]   'GRAPH IS CONNECTED
[35]   →0
[36]  L:'GRAPH IS DISCONNECTED
[37]   →0
       ∇
```

Par. 4.

Some examples of the use of ALGOL68.

In connection with par. 3 we can try to make a program to calculate the connectivity of a graph in Algol68.
In fact our program will do the same as the function COMP in APL.
We make use of the procedures COMPR and MEMB of par. 2 and the procedure MEMBC and TRANSF

(a) TRANSF is a procedure which transfers the matrix ZA (see par. 3) in a special Algol68 structure called MATR, this has to be coded in the program with a so called mode declaration:

```
'MODE 'MATR='STRUCT((1'UPB ZA)'REF()
'INT ADL, (1:1'UPB ZA)'INT IN);
```

This structure consists of an adjacency list of ZA (called ADL) which has the type 'ROW 'OF 'ROW and an integer array (called IN) which tells us, how many elements there are in each row of ADL. The coding of TRANSF has been put in appendix B

(b) In the program it is useful to have a procedure MEMBC which looks like the procedure MEMB, the input parameters are the structure MATR from above (it is possible to use MATR as input parameter), a row of integers R and an integer Q. MEMBC makes use of MEMB. memb. The result is a boolean. MEMBC examines if the row Q of the ADL part of MATR has something in common with a certain row of integers R.
The text of MEMBC follows:

```
'PROC MEMBC=( 'MATR P,( ) 'INT R ,'INT Q)
'BOOL:
'BEGIN 'BOOL FOUND:=FALSE;
'LOC(1: (IN 'OF P)(Q)) 'INT PN;
 'FOR I 'TO (IN 'OF P)(Q) 'DO
 PN(I):=(ADL 'OF P)(Q)(I) 'OD;
'FOR I 'FROM 'LWB R 'TO 'UPB R
 'WHILE 'NOT FOUND
'DO (MEMB(PN,R(I))='TRUE/FOUND:='TRUE)
 'OD; FOUND 'END;
```

Now the coding of the main program calling TRANSF, MEMBC and other procedures follows. The input to the program is certain matrix ZA, as in par. 3. Comments have been placed between 'CO and 'CO.

```
'CO FIRST THE DECLARATIONS 'CO
'LOC (1:9,1:9) 'INT ZA; 'INT Q,J;
'LOC (1:40) 'INT COLL; 'LOC (1:80) 'INT
TOTAL;
'CO THE ADJACENCY LISTS ARE PUT TOGETHER IN
TOTAL 'CO
'CO THE LENGTH OF THE ARRAYS COLL AND TOTAL
ARE NOT QUITE PREDICTABLE 'CO
'MATR M1:=TRANSF(ZA);
'FOR I 'TO 40 'DO COLL (I):=0 'OD;
'FOR I 'TO 80 'DO TOTAL(I):=0 'OD;
'INT T:=1; 'INT COLLC:=0; 'INT CNT:=0;
'INT VL:=0; 'INT M:=1;
TOTAL(1):=1;
'CO WE NEED THE COUNTERS COLLC AND CNT TO
KNOW HOW MANY 'CO
'CO ELEMENTS OF THESE ARRAYS HAVE BEEN FILLED
WITH NON ZERO 'CO
PART1:
'CO TRY TO ADD SOMETHING TO TOTAL 'CO
Q:=TOTAL (M);
     'IF MEMB(COLL,Q)='TRUE 'THEN 'GOTO LL
     'FI; 'FOR I 'TO (IN 'OF M1)(Q) 'DO
'CO SEE REMARK 1 'CO
     TOTC=TOTC+1;
     TOTAL(TOT):=(ADL 'OF M1)(Q)(I) 'OD;
'CO SEE REMARK 1 'CO
     COLLC:=COLLC+1;
     COLL(COLLC):=Q;
LL:  M:=M+1;
     'IF M<(T+1) 'THEN 'GOTO PART1 'FI;
     and so on ...
```

Further comments.

Remark 1.

With the fieldselections ADL 'OF and IN 'OF the two parts of the structure 'MATR M1 can be used in the program.

Remark 2.

As you see, a number of loop clauses and conditional clauses have been used in the program.

Remark 3.

In APL there are primitive functions for membership and simple operations to get rid of duplicates out of a row. You can easily increase the length of a vector by catenation. In Algol68 you should declare enough space in advance.

Conclusion.

In this paper main features of APL and Algol68 have been considered.
APL is a nice language with many possibilities and easy to learn. There are many applications in the field administration (databases), graph steory and other areas of mathematics.
In numerical mathematics applications in the field of series and polynomials are well known (see reference 2).
In administration it can be used for all kinds of data-analysis and reduction including report writing.
In graph theory the use of matrices with ones and zeros give many possibilities to use primitive functions.

Other large iterative calculations might be possible but may give performance problems.
Two disadvantages of APL are well known, the lack of control structures and the restrictions in the use of functions. Functions with more than two parameters are not permitted.
Algol68 is a batch language with many tools, it takes much more time to learn than APL.
If small reports are necessary in an environment of industry or administration one should prefer APL. For applications in the sense of par. 3 and par 4, you can see that the use of APL gives quick results, you don't have to construct so many subroutines as in Algol68.
Algol68 is not better than APL if small calculations in the field of numerical analysis are required or in special areas as calculations with series and polynomials.

For complex calculations Algol68 might be better than APL. Also in the case of complicated datastructures Algol68 is favorite. It has many tools, and the programmer can construct his own as needed.
However, it is a question if programs with complicated datastructures are efficient.

References.

(1) C.H. Lindsey, S.G. van der Meulen; Informal Introduction to Algol68. North-Holland Publishing Company 1977.
(2) P.L.J. Siero; Het gebruik van polynomen en reeksen in APL; CRI-bulletin, januari 1978.

Appendix A.

```
      ∇EULER[⎕]∇
      ∇ R←V EULER ZA;B;C;N;CIRC;M;A;O
[1]   ⍝ ZA IS A CONNECTED GRAPH
[2]   ⍝ V IS THE STARTING POINT
[3]   ⍝ MATRIX M IS WORKAREA
[4]    O←40⍴1
[5]    M← 10 40 ⍴0
[6]    N←1
[7]    A←1↑N
[8]   ⍝TRANSFER FROM ZA TO  VECTOR A
[9]   ⍝ FOR COMPACT NOTATION
[10]  LOOP:
[11]   B←N+(N↓ZA[;N])/⍳⍴(N↓ZA[;N])
[12]   C←((2×⍴B)⍴ 1 0)\B
[13]   C[2×⍳⍴B]←N
[14]   A←A,C
[15]   →((N←N+1)≤1↑⍴ZA)/LOOP
[16]  ⍝ LOOK FOR CIRCUITS IN PART1
[17]   CIRC←,1
[18]   C←A⍳1
[19]   B←0
[20]  PART1:→((2|C)=0)/EVEN
[21]   N←A[C+1]
[22]   A←(((C-1)↑O),0,0,((⍴A)-(C+1))↑O)/A
[23]   →L
[24]  EVEN:N←A[C-1]
[25]   A←(((C-2)↑O),0,0,((⍴A)-C)↑O)/A
[26]  ⍝ A GETTING SMALLER UNTIL ⍴A=0
[27]  L:→(N=CIRC[1])/ADD
[28]   CIRC←CIRC,N
[29]   C←A⍳N
[30]   →PART1
[31]  ADD:M[(B←B+1);]←40↑CIRC
[32]  ⍝SAVE THE CIRCUIT IN MATR
[33]   →((⍴A)=0)/PART2
[34]   C←1
[35]   CIRC←1↑A
[36]   →PART1
[37]  PART2:R←((B,40)↑M) PATH V
[38]  ⍝  CALL FOR   FUNCTION  PATH
[39]   →0
      ∇
```

```
      ∇PATH[⎕]∇
      ∇ R←MATR PATH V;I;HELP;U;W
[1]   ⍝INITIALLY MATR CONTAINS ALL
[2]   ⍝THE CIRCUITS OF THE GRAPH
[3]   ⍝PAR 1:ROWS OF MATR,THAT
[4]   ⍝HAVE NOT BEEN USED UNTIL NOW
[5]   ⍝PAR 2:IN THE BEGIN START VERTEX,
[6]   ⍝LATER OUTPUT OF LAST CALL OF PATH
[7]   ⍝OUTPUT:EXPANSION OF  R(V),WITH
[8]   ⍝NEW CIRCUITS CONTAINING R(V)
[9]    R←,V
[10]   I←0
[11]  START:→((I←I+1)>1↑⍴MATR)/END
[12]   →(MATR[I;1]=0)/START
[13]   W←(MATR[I;]∊V)
[14]   →((+/W)=0)/START
[15]  ⍝R  GETS LONGER IF R AND
[16]  ⍝MATR[I;] HAVE COMMON ELEMENTS
[17]  ⍝U IS THE SMALLEST ELEMENT OF
[18]  ⍝MATR[I;],THAT BELONGS TO R
[19]   U←W⍳1
[20]   W←R⍳(MATR[I;U])
[21]   HELP←(MATR[I;]≠0)/MATR[I;]
[22]  ⍝HELP IS ROTATED UNTIL U IS THE
[23]  ⍝LAST ELEMENT,THE RESULT IS IN U
[24]   U←(U-⍴HELP)⌽HELP
[25]   R←((W↑R),U,(W↓R))
[26]  ⍝NEW VERTICES ARE ADDED
[27]  ⍝TO EARLIER RESULTS
[28]   MATR[I;1]←0
[29]   →START
[30]  END:W←W/⍳⍴(W←MATR[;1]≠0)
[31]   →((1↑⍴(MATR←MATR[W;]))=0)/STOP
[32]   R←MATR PATH R
[33]  STOP:→0
      ∇
```

Appendix B.

```
'PROC TRANSF=((,)'INT Y)'MATR:
'BEGIN
'LOC 'MATR M;
'LOC (1:1'UPB Y)'INT INF;
'LOC (1:1'UPB Y)(1:10)'INT INTR;
'FOR N 'FROM 1 'TO 1'UPB Y 'DO
'LOC 'INT J:=0;
'LOC (1:1'UPB Y)'INT PN:=Y(,N);
'FOR I 'FROM (N+1) 'TO 'UPB PN 'DO
'IF PN(I)¬=0 'THEN
J:=J+1; INTR(N)(J):=I 'FI      'OD;
INF(N):=J                      'OD;
'LOC (1'UPB Y)'REF( )'INT W;
'FOR N 'TO 1'UPB Y        'DO;
W(N):='LOC (1:INF(N))'INT 'OD;
'FOR N 'TO 1'UPB Y  'DO
'FOR J 'TO INF(N)   'DO
W(N)(J):=INTR(N)(J) 'OD
                    'OD;
IN 'OF M:=INF;
ADL 'OF M:=W;
M
'END;
```