A VECTOR LANGUAGE FOR THE SOLUTION OF PDE PROBLEMS

by

John Gary
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-068-75

April 1975
Rewritten September 1976

This research was supported by NSF grant ATM75-17036.

1. Introduction. We are mainly concerned with the inclusion of a vector capability within a higher level language used to construct codes for the solution of partial differential equations. We attempt to design the vector constructs so that efficient programs can be compiled on a parallel computer such as the Texas Instruments ASC, the Cray computer, or a new version of the Illiac IV. The language should be designed so that it is easy for the user to differentiate between those constructs which can be compiled efficiently and those which can not. We should also impose restrictions so that implementation of an efficient compiler is not too difficult. The language should be designed so that it can be implemented on a serial computer by a preprocessor which generates an object FORTRAN program.

2. The vector structure. We hope to provide a vector structure which has some of the power of APL but is tuned more closely to finite difference programs for partial differential equations. Our vectors are created by means of an "INDEX" applied to an array. The index gives the lower and upper bounds for a particular subscript of the array. For example,

```
I,J : INDEX
VA,VB : ARRAY [-40..40] OF REAL
AA,AB : ARRAY [1..40,1..40] OF REAL
I = 3..38
J = 1..40
AA[I,J] = VA[I]*(AA[I+2,J]-AA[I-2,J]) + 3.*VB[I]
I = 1..40
AB[I,J] = .5*(AA[I,J]-AA[J,I])
```

The operations are done componentwise, associating components of the indexed array which have the same symbolic index. Thus AB is the skew symmetric part of AA. It is assumed that the operations are done simultaneously for all components of the indexed expressions. Thus the expression

```
I = 2..40
VA[I] = VA[I-1]
```

is the same as the FORTRAN computation

```
       DO 10 I = 2,40
    10 VB[I] = VA[I-1]
       DO 20 I = 2,40
    20 VA[I] = VB[I]
```

We extend this to allow array slices similar to ALGOL 68, however with more general operations similar to APL. Here the operations are on conformable vectors as in APL. In this case no symbolic subscript indices are used to define the operations, instead the subscript ranges are given explicitly. For example

```
    AA[3..6,7] = AA[1..4,1] + AA[7,3..6]
```

Note that

```
    AA[1..10,1..10] = AA[1..10,1.10]*VA[1..10]
```

is not allowed since the rank two vector AA[1.10,1..10] is not conformable with VA[1..10]

We do not propose to allow general vectors of the sort allowed in APL. We would not allow for example the APL vector subscript

```
    VA[1,-3,7,9]
```

However we would allow

```
    VB = (1.,-49.,17.,21.)
```

The vector VA requires irregular memory accessing which can not be done efficiently on a parallel computer. The vector VB is defined dynamically which we allow in a less general form than APL. The generality of APL might require an interpreter rather than a compiler, or a very complex compiler.

We have used a syntax similar to PASCAL except that an end-of-line is also an end-of-statement unless the line terminates with ";+". The character ";" is a statement separator along with the end-of-line. We will define our vector structure more completely in the following sections.

3. Indexed vector expressions. Our primary means of generating vector operations is by use of an index in an array subscript. This is somewhat similar to the INC_SET used in the TRANQUIL language [4]. The arrays must be declared in constrast to APL, however the dimension of the array may be determined dynamically. An index variable is declared as follows:

```
ID,JD,RD : INDEX
```

It sets a lower and upper bound for a subscript. Such a variable can be set to an indexed expression which consists of two integer valued expressions separated by the double period delimiter "..". For example

```
ID = 2..(N-1)
```

The only operation which applies to index variables is the addition (or subtraction) of a scaler integer. For example

```
ID = ID+21
ID = ID+3*(N-1)
```

These index variables can be used as array subscripts. For example

```
A,B : ARRAY [1..100,1..2] OF REAL
ID,JD : INDEX
ID = 2..99
JD = 1..2
A[ID,JD] = (A[ID+1,JD] - A[ID-1,JD])/DLX
```

The FORTRAN equivalent of the last line is

```
    DO 10 JD = 1,2
    DO 10 ID = 2,99
10  TMP(ID,JD) = (A(ID+1,JD) - A(ID-1,JD))/DLX
    DO 11 JD = 1,2
    DO 11 ID = 2,99
11  A(ID,JD) = TMP(ID,JD)
```

The array declarations may be dynamic, defined by a subroutine argument or by a specific allocation command. For example

```
A : ARRAY [   ] OF REAL
N : INTEGER
RA : ARRAY [2,3] of INTEGER
FOR L = 1 TO 3
  RA(1,L] = -10
  RA[2,L] = 10
ENDFOR
ALLOCATE (A,3,RA)
```

This creates an array A of rank 3 whose dimension is determined by RA, namely A[-10..10,-10..10,-10..10]. a DEALLOCATE instruction is also included. A run time "heap" whose size is controlled by the user must be provided. Note that the dimension of an array A is an array $\{D_{\ell k}\}$ $1 \le \ell \le 2$ and $1 \le k \le$ rank (A). The rank is the number of subscripts. Dynamic arrays must be declared with a blank subscript range, that is a blank dimension.

Index expressions can be used as subscripts. These are of the form

```
<index> ± <scaler integer expression>
```

For example

```
ID,JD : INDEX
N,M : INTEGER
A : ARRAY [1..100,2] OF REAL
A[JD,1] = 0.
A[ID-M,2] = A[ID,1]
```

The index expressions are JD,ID-M, and ID.

These index expressions, used as array subscripts, create what we will term indexed vector expressions. This is our basic vector structure. For example

```
ID,JD : INDEX
A,B : ARRAY [1..100,1..100] OF REAL
U,V,W : ARRAY [1..100,2] OF REAL
S,T : ARRAY [1..100] OF REAL
C1,C2,C3 : REAL
N1,N2,N3 : INTEGER
```

```
ID = 1..100
JD = 2..99
N1 = 1
N2 = 2
U[JD,N2] = C1*(U[JD+1,N1] - U[JD-1,N1])*V[JD,N1] + S[JD]
A[ID,JD] = B[ID,1]*U[ID,2] + W[JD,2]
```

Each indexed array can be regarded as a new array whose subscript limits are determined by the set of index variables used in the subscripts of the array reference. We call this set the index list. We regard this new array as an "indexed vector expression". For example the index list of U[JD,N2] and U[JD+1,N1] is JD alone. Indexed vector expressions may be combined by the standard arithmetic operators to yield new expressions. The index set of the result is the union of the index sets of the operands. For example, the FORTRAN equivalent of the last line in the above example is

```
    DO 10 JD = 2,99
    DO 10 ID = 2,99
10  A(ID,JD) = B(ID,1)*U(ID,2)+W(JD,2)
```

The index set of B[ID,1]*U[ID,2] is ID while that of B[ID,1]*U[ID,2]+W[JD,2] is union (ID,JD) .

For reasons which we state in section 7, we wish to have this index list ordered. There seems to be no very natural ordering of the index list of an expression, as the following example shows.

```
A[I,J] + B[J,I]
```

However, we will order the indices of an indexed array by the order of their appearance and then order the result of an operation by taking the order of the first operand and then adding any indices which appear in the second operand but not the first in the order of appearance in the second. In finite difference codes there is not likely to be any difficulty with this ordering. As an example consider

```
I,J,K : INDEX
(A[I,J] + B[J,K,I])*C[K,I]
```

The ordered index list of this expression is (I,J,K).

This type of vector definition is intended for finite difference expressions over a mesh which one typically writes in the form

$(u_{i,j+1}-u_{i,j-1})*(c_{j+1}+c_{j-1})/(4\Delta y)$. The subscripts give a symbolic representation of the mesh points. Note the difference between these vectors and those of APL. The expression

```
I←ιN
J←ιN
U[I;J]←V[I;J]*C[J]
```

is not defined in APL because the vectors V[I;J] and C[J] are not conformable. However the following expression, which is common in finite difference codes, is defined as an indexed vector expression

```
I = 1..N
J = 1..N
U[I,J] = V[I,J]*C[J]
```

Another difference is illustrated by

```
I←ιN
J←ιN
A[I;J]←(B[I;J]-B[J;I])/2.
```

In APL, we have A[I;J] = (B[ιN;ιN]-B[ιN;ιN])/2=0. As an indexed vector expression

```
(B[I,J]-B[J,I])/2.
```

is the skew symmetric part of B. In APL the direct product

```
A ← U[ιN] o.* V[ιN]
```

yields the same rank two array as the following indexed vector expression

```
I = 1..N
J = 1..N
A[I,J] = U[I]*V[J]
```

Note that the index list of the indexed array on the left of a replacement must contain the index list of the expression on the right. The first replacement is valid

```
I = 1..N
J = 1..N
A[I,J] = U[J]
```

however the second is not.

```
U[I] = A[I,J]
```

The basic arithmetic operations allowed between indexed expressions are +, -, *, /, and **. We give them the same operator precedence that they have in FORTRAN, unlike APL.

We have used indexed or increment subscripts similar to those proposed for the TRANQUIL language [4]. Another possibility is the use of logical vectors of rank one whose dimension (or length) is compatible with the array subscript. For example.

```
IL,JL : ARRAY [-50..50] OF BOOLEAN
A,B : ARRAY [-50..50,-50,50] OF REAL
IL = 1
IL = B[IL,1] > 0.
JL = 1
JL = B[1,JL] > 0.
A[IL,JL] = SQRT(B[IL,1])*SQRT(B[1,JL])
```

Here the operation is carried out only for those values of IL and JL which are equal to 1 (i.e., "TRUE"). We prefer the incremented index variables because a vector language based on such variables is perhaps easier to implement efficiently on a variety of machines. Also the indexed arrays seem to be more convenient for finite difference codes.

4. <u>Conditional evaluation of indexed vector expressions</u>. This is implemented by means of the INDEXIF, IFALL, and IFANY compound statements. For example

```
I = 1..N
IFANY (U[I] "EQ" 0.) THEN
    UFLAG = 1
ELSE
    UFLAG = 0
ENDIF
J = 1..N
INDEXIF(X[I]*Y[J] "NE" 0.)
    A[I,J] = 1./X[I]+1./Y[J]
```

```
ELSE
    A[I,J] = 1.E+30
ENDIF
```

In the case of the INDEXIF the truth portion of the statement is executed only for those values of the index list (I,J) for which the Boolean is true. In the range of the INDEXIF we will allow only replacement statements whose left side index list equals the index list of the Boolean expression. We will also allow other INDEXIF and IFANY, a IFALL statements provided the index list of their Boolean expression equals the index list of the containing indexed IF statement. Within the scope of an IFANY or IFALL the index list restriction of containing index IF statements is effectively suppressed since these are scale statements. The active elements of nested index lists are obtained by an "and" of the active elements in the nested IF statements.

The relational and Boolean operators can be denoted by

```
<, >, <=, >=, ==, =/
```

or possibly

```
"LT", "GT", "LE", "GE", "EQ", "NE"
```

or

```
"LT, "GT, "LE, "GE, "EQ, "NE
```

The Boolean operators are similar. The language should require only characters found on most timeshared terminals. The syntax treats a string of blanks as an end marker for a token. We will use the FORTRAN operator precedence for these operators.

<u>5. Additional operations</u>. The APL reduction and inner product operations can be combined with the indexed vector expressions in a reasonable way. Perhaps the only reduction that is required is summation which is denoted by +! . For example

```
I = 1..N
SX = +!X[I]
```

This is equivalent to the Fortran program

```
          SX = 0.
          DO 10 I = 1,N
      10  SX = SX+X(I)
```

Reduction over specific indices can be indicated by giving the indices immediately following the reduction operator otherwise the reduction is done over all indices. For example

```
      I = 1..N
      J = I
      SXI = +![I]A[I,J]
```

Note that an inner product, for example a product of a matrix with a vector, can be denoted in this manner. For example

```
      I = 1..N
      J = 1..N
      Y[I] = +![J](A[I,J]*X[J])
```

The inner product involving addition and multiplication is denoted by +!*. In this case the reduction by summation occurs over all indices which are common to the two operand expressions. Thus the multiplication of two matrices is given by

```
      I = 1..N ; J = 1..N ; K = 1..N
      A[I,J] = B[I,K]+!*C[K,J]
```

If the reduction is to be carried out only over specified indices, then these indices can be given immediately following the inner product operator. For example

```
      I = 1..N ; J = 1..N
      X[I] = A[I,J]+!*[J]B[I,J]
```

Various functions can be defined for indexed vector expressions. For example "RANK" is a function with an integer value, "DIMEN" has a vector value, and "VMAX" and "VMIN" have a real value.

```
A : ARRAY [-50..50,-5..5] OF REAL
I = 0..50 ; J = -2..2
N1 = "RANK"(A[I,3])
N2 = "RANK"(A[I,J])
V = "DIMEN"(A[I,J])
```

Here

$$N1 = 1$$
$$N2 = 2$$
$$V = \begin{vmatrix} 0 & -2 \\ 50 & 2 \end{vmatrix}$$

The reduction "+!" will have the same priority as + and likewise for other reductions (when they are included). The inner product +!* should have the same priorty as "*".

6. Control structure. A reasonable set of control statements are included. These are illustrated by the following.

```
FOR I = <exp1> to <exp2> BY <exp3>
  .
  .
  .
ENDFOR

REPEAT
  .
  .
  .
UNTIL <Bool exp>

WHILE <Bool exp>
  .
  .
  .
ENDWHILE

LOOP
  .
  .
EXITIF <Bool exp> TO <label>
  .
  .
ENDLOOP
```

```
IF <Bool exp1> THEN
  .
  .
  .
ORIF <Bool expn>
  .
  .
ELSE
  .
  .
ENDIF

SELECT <exp> FROM
CASE <exp1>
  .
  .
  .
CASE <expn>
  .
  .
  .
OTHERWISE
  .
  .
ENDSELECT
```

The only "GOTO" statement which is included is the following.

```
EXITTO <label>
```

This permits a transfer to the first statement following the end of a block containing the EXITTO. Only the end-of-block statements can be labeled (that is ENDIF, ENDWHILE, ENDFOR, ect.). A block can also be designated by the BEGIN...END statements. The only purpose of BEGIN...END is to provide labeled exit points for the EXITTO statement since the compound conditional statements have a unique end-marker. We feel this design is less error prone than the use of BEGIN...END along with ";".

7. General vectors. In addition to indexed vector expressions, we will allow a more general vector type similar to that in APL. This may prove to be too difficult to implement, but we feel it is worth a try. We believe that it is easier to compile and read the program

if the vectors are denoted by a specific delimiter rather than being the result of concatination by blanks or commas. The symbol "//" is used to delimit a vector of rank one. For example

```
          V = //0.,-5.,EXP(3.),1+2.*x,17.,x**2//
          RW = "SHAPE(//2,3//,V)
```

The function "SHAPE (or "SHAPE") creates a vector whose rank is the dimension of the rank one vector which is the left argument. The vector is formed from the elements of the right argument which must be a scaler or a vector of rank one. If there are too few elements in the right argument, then this argument is used repeatedly. In order to be compatible with Fortran, the vectors are ennumerated by columns rather than the row ennumeration used in most languages. The function is taken from APL. We use a named function rather than a binary operator because we do not wish to be restricted to terminals which have the APL character set. Vectors can also be formed from arrays, however general vector subscripts are not allowed, contrary to APL. A blank subscript implies the entire extent of that array subscript is used to form the vector. The dimension of a dynamic array (that is, one with a blank dimension declaration) can be set by the ALLOCATE statement or by the use of the array name on the left side of a replacement statement. In this case the array takes the dimension of the right side expression. Two vector expressions are conformable if the length of corresponding subscripts is the same. A scaler subscript (a subscript of length one) is ignored. Thus the following are conformable vectors.

```
          A[2..3,7,1..4] and B[1..2,2..5]
          X[-50..50] and Y[1..101]
```

The rank of a scaler expression is not defined. The rank of a vector, to which space is not yet allocated, is zero. The following vector is not allowed since the subscript is not an index expression.

```
          V(//1,7,2,3//)
```

All vectors formed with general index expressions, which do not involve an INDEX variable and are thus not "indexed vector expressions", are taken with subscript range starting at one. Thus we have

$$\text{"DIMEN(A[-2..2,2,2..4])} \equiv \begin{vmatrix} 1 & 1 \\ 5 & 3 \end{vmatrix}$$

That is,

```
"DIMEN(A[-2..2,2,2..4])[2,1] ≡ 5
```

This is done because there seems to be no reasonable way to assign a subscript range to an expression such as

```
U[1..5]+V[-2..5]+//0.,-1.,1.,-2.,2.//
```

We need a way to "coerce" an indexed vector expression into a general vector. This is the reason why we insisted earlier that the index list associated with an indexed vector expression be ordered. Then we may regard an indexed vector expression as an array, taking the subscript limits of the array to be those of the members of the index list taken in order. Consider the following example.

```
ID,JD : INDEX
U,V : ARRAY [-50..50,5] OF REAL
A,B : ARRAY [  ] OF REAL
ID = 0..50
JD = 3..5
A = U[ID-50,1]+U[ID,1]
B = V[ID,JD]*V[ID,JD-2]
```

In this example we have, as a result of the replacement statements which define the dimension of A and B

$$\text{"RANK(A)} \equiv 1$$

$$\text{"DIMEN(A)} \equiv \text{//0,50//} \equiv \begin{vmatrix} 0 \\ 50 \end{vmatrix}$$

$$\text{"RANK(B)} \equiv 2$$

$$\text{"DIMEN(B)} \equiv \begin{vmatrix} 0 & 3 \\ 50 & 5 \end{vmatrix}$$

We will need an outer product for these more general vectors. It is not defined for indexed vector expressions. This outer product is *.! . For example, if

```
A = //1.,2.,3.//*.!//-1.,1.//
```

then

"DIMEN(A) = $\begin{vmatrix} 1 & 1 \\ 3 & 2 \end{vmatrix}$

A = $\begin{vmatrix} -1. & 1 \\ -2. & 2 \\ -3. & 3 \end{vmatrix}$

A[2,1] = -2.   A[3,2] = 3.

Other operators should be allowed in the outer product, for example >.! or =1.! . The result is of type logical which is regarded as a zero or one in order to mix logical and arithmetic types.

One reason for the inclusion of these general vectors is to permit the use of the APL compress and expand operators. These operators do not work well with indexed vector expressions. We will define these operators as functions of two arguments rather than operators since we prefer to spell out the operators rather than giving a new symbol for them. Also we restrict these functions to vectors of rank one. For example

```
LV : ARRAY [1..100] OF BOOLEAN
X,Y,Z : ARRAY [1..100] OF REAL
LV = X>0.
Y  = "COMPRESS(LV,X)
```

In this case Y is a vector of dimension 1..N where N is the number of positive elements in X. If N is zero, then Y is not defined. Undefined vectors have rank zero and the "DIMEN function is not defined. The expand function is also included, for example

```
Z  = "EXPAND(LV,Y)
```

Also a "MERGE function is included. It has three arguments, the first logical, the second two of the same arithmetic type. All three must be conformable. For example

```
Z = "MERGE(LV,X,Y)
```

If LV[I] = 1, then Z[I] = X otherwise Z[I] = Y[I] (I is not an index).

<u>8. Linkage to subprograms.</u> The language must permit separately compiled subprograms, otherwise it is difficult to bring up a large code on a time shared "code development" computer. For example, one may wish to bring up a large code on a PDP-10 or even a PDP-11 with the intention of running production on a Cray machine. It is not

advisable to recompile the entire code on the smaller machine each time a change is made. We find it effective to run tests for a finite difference code on a small grid (5x5, or 5x5x4 for example) using the code development machine before going to the production machine. Gannon [9] has presented some ideas and experiments concerning the reliable linkage of procedures, and we have tried to incorporate some of these into our language design.

We require a "preamble" which gives a complete definition of the argument list for each subprogram which is called or defined in the subsequent code. Parameters used only for input can be so designated by use of the INPUT mode. Actual parameters which are expressions must correspond to formal parameters of INPUT mode. We regard indexed array references as arrays whose subscript limits are determined by the subscript index expressions. For example consider the declaration

```
S1 : DECLARE SUBROUTINE
       INPUT(X : ARRAY [  ] OF REAL)
       OUTPUT(EX : ARRAY [  ] OF REAL,IERR : INTEGER)
     ENDDECLARE
```

The subroutine body might be defined by

```
S1 : SUBROUTINE
       INPUT(X : ARRAY [  ] OF REAL)
       OUTPUT(EX : ARRAY [  ] OF REAL,IERR : INTEGER)
       XN : ARRAY [  ] OF REAL
       EPS : REAL
       N,NMAX : INTEGER
       CONSTANT EPS = 1.E-6,NMAX = 200
       BEGIN
          XN = X
          "ALLOCATE(EX,"DIMEN(X)) ; EX = 1.
          N = 1
          LOOP
             EX = EX+XN
             N = N+1
             XN = XN*X/N
             EXITIF("VMAX"(XN)<EPS) TO DONE
             EXITIF(N>NMAX) TO FAILURE
```

```
                ENDLOOP
                   IERR = 0
                   RETURN
                END : FAILURE
                   IERR = 1
                   RETURN
          ENDSUBROUTINE
```

The subroutine call might be

```
          X : ARRAY [-50..50] OF REAL
          EX : ARRAY [0..100,2] OF REAL
          N = 10
          I = -N..N
          CALL S1(X[I],EX[I+N],IERR)
```

A function is allowed to have an array value. The dimension of this returned array may be determined dynamically. For example

```
          F1 : FUNCTION OF ARRAY [   ] OF REAL
               INPUT(X,Y : ARRAY [   ] OF REAL)
               LV : ARRAY [   ] OF BOOLEAN
               LV = X<Y
               F1 = "MERGE(LV,Y,X)
               RETURN
               ENDFUNCTION
```

The call F1(X,Y) is equivalent to "MERGE(X<Y,Y,X)

Subroutine linkage can also be accomplished by means of COMMON. The COMMON declarations must also be placed in the preamble. In order to use COMMON variables in a subroutine, a USECOMMON statement must be placed in the subroutine giving the names of the variables to be used. The USECOMMON statement will also indicate if the variables are used as input only (INPUT) or as output or both input and output (OUTPUT). This is in accordance with the suggestions by Gannon [9]. For example, we have the declaration.

```
          COMMON BLK1
               X,Y,Z : REAL
          COMMON BLK2
               A : ARRAY [-20..20,1..10] OF REAL
               N,M : INTEGER
          ENDCOMMON
```

Usage of A and M in a subroutine would require the following declaration within the subroutine

```
          USECOMMON BLK2
               INPUT(M)
               OUTPUT(A)
          ENDUSE
```

9. I/O Commands. In addition to the I/O commands found in Fortran, there are several features which would be useful in finite difference calculations. These include format-free print statements which handle vectors and provide labeled output. Convenient graphics output would be extremely useful as well as buffered I/O, both random access and sequential. File declarations which can be of record type as in PASCAL would be desirable.

We will describe only one type of I/O structure which is intended to facilitate problems in which the arrays must be buffered in from a disk. What we need is an array of records in the PASCAL sense, however we wish to address this data in a way which is convenient for both I/O and mesh calculations. We suppose that we have five lines of a mesh in an array in the memory. At the same time that we are computing with the data in these five lines we are reading in another line from the disk and writing one of the lines out to the disk. Therefore we want a BLOCK structure such as the following.

```
          BK : BLOCK [1..6]
               U,V,W,T : ARRAY [1..80,1..40] OF REAL
                    GH : ARRAY [1..80] OF REAL
               ENDBLOCK
```

We wish to refer to the arrays in block as arrays of rank three, that is U[I,J,K] where $1 \le I \le 80$, $1 \le J \le 40$, $1 \le K \le 6$. In addition we want to issue

I/O commands which refer to an entire line, that is BK[K] where $1 \le K \le 6$. Therefore each of the lines should be stored contiguously in the memory so that data can be read directly into the block without first being read into a buffer.

10. Language specification. We have not given a precise description of either the syntax or semantics of our proposed language. We have not made a decision on many questions and have surely overlooked many others. For example, should we allow only explicit definition of dynamic array dimensions with the ALLOCATE, or should we permit an array dimension to be defined or altered by appearance of the array name on the left side of a replacement statement? The latter is convenient, but somewhat inefficient and perhaps error prone.

We will try to implement, on the Cray machine, a subset of the vector arithmetic features of the language in the near future. Our objective is to determine if single statement optimization can produce reasonably good code. We hope to gain some insight into the difficulty of applying standard optimization techniques to our vector expresssions.

REFERENCES

[1] N. Wirth (1972), "The Programming Language PASCAL", Eidgenönische Technische Hochschule Zurich.

[2] IVTRAN (1973), "The IVTRAN Manual", Massachusetts Computer Associates, Wakefield, Massachusetts, 01880.

[3] D. Lawrie (1973), "Memory-Processor Connection Networks", Ph.D. thesis, University of Illinois, Comp. Sci. Rep. 557.

[4] P. Budnik and D. Kuck (1969), "A TRANQUIL Programming Primer", Department of Computer Science, Rep. 816, University of Illinois.

[5] H. Katzan (1970), "APL Programming and Computer Techniques", Van Nostrand, New York.

[6] Löcs and Gary (1974), "A FORTRAN Extension for Data Display", IEEE Trans. on Comp., 1257-1263.

[7] M. Wilson, "Flexible Subarray Facilities for Classical Programming Languages", IBM Houston Scientific Center Technical Report No. 320-2426, IBM Corp., 6900 Fannin Street, Houstin, Texas, 77025.

[8] A. Haberman (1973), "Critical Comments on the Programming Language PASCAL", Acta Informatica, 3, 47-57.

[9] J. Gannon (1975), "Language Design to Enhance Programming Reliability", Rep. CSRG-47, Comp. Sys. Research, University of Toronto.