A. Hassitt
L. E. Lyon

Efficient Evaluation of Array Subscripts of Arrays

Abstract: The APL language allows subscripted expressions such as $A[I;J;K]$, where A is an array and I, J and K may be scalars, vectors or arrays of any size and shape. We describe an efficient method of evaluating these expressions. The method is quite general and yet it does recognize all the special cases in which multiple subscripts can be reduced to a simpler form. We show how the same mechanism can be used to evaluate other selection operations, such as transpose, and how selection operations may be combined.

Introduction

When programs written in a high-level language such as FORTRAN or PL/1 are run on the computer, they usually go through three stages, namely compilation, program loading, and execution. At each of these stages certain decisions about data structure and data representation have to be made. In the early languages such as FORTRAN 1, all of these decisions could be made at the compiler stage. In more modern languages such as PL/1, the compiler can determine most of the structure and the data representation, but some of the decisions have to be deferred until execution time; for example, the memory address of an automatic variable is not fixed until the procedure that uses the variable is executed. The advantage of making an early decision about structure is that the compiler can then try to optimize the object program. The advantage of making a late decision, on the other hand, is that the programming language can be more powerful and easier to use; in certain cases the late decision can lead to a more optimum use of the computer.

APL\360[1] removes many of the restrictions of previous languages; it allows the structure and the representation of a variable to change at any time during execution. APL is a powerful and elegant language (see for example, Kolsky[2]) but the construction of an efficient APL interpreter does lead to some complex problems. In this paper we are particularly concerned with the evaluation of APL subscript expressions. At the present time, this problem is peculiar to APL, but as new and more powerful languages are developed we would expect this type of problem to arise in many other situations.

Specification of the problem

Consider first the problem of subscripts in FORTRAN. If the FORTRAN programmer writes a statement such as Z = A(I,J,K), then A must be a three-dimensional array, I, J and K must be scalars, and the dimensions of A must be either constants or simple scalar variables. It is comparatively easy to generate code which will extract the relevant element from A and put it in Z. Although the subscript evaluation is quite straightforward, it is somewhat more difficult to produce optimum code. An optimizing compiler will evaluate the constant parts of the subscript at compile time, will try to replace any multiplications by additions, and will move parts of the subscript evaluation outside any of the enclosing DO loops, see Allen[3], Hopgood[4], Busam and Edlund[5], and Lowry and Medlock[6].

PL/1 allows scalar subscripts and it also allows statements such as Z = A(I,*,K). In this case, if A has dimensions N1, N2, N3, then Z must be a vector of N2 elements and the statement sets Z(J) = A(I,J,K) for all valid values of J. The compiler has to produce the code to perform

```
DOJ = 1 TO N2;      Z(J) = A(I,J,K);      END;
```

and this is not difficult to do.

Before discussing APL subscripts, let us review some of the terms used in APL. If A is any variable, then the *size* of A is a vector which gives the dimensions of A. The *rank* of A is the size of the size of A; it gives the number of dimensions. If A is a 5×6 matrix, then the size of A is 5 6, and the rank is 2. The *ravel* of A is a vector which

contains all the elements of A. If A is a scalar, then the ravel of A is a one-element vector. If A is a vector, then the ravel of A is the same as A. If A is, for example, a 5×6 matrix, then the ravel of A is a vector of 30 elements with values $A[0; 0], A[0; 1], \cdots A[0; 5], A[1; 0], \cdots A[4; 5]$. APL subscripts can begin at either "0" or "1", in this example, and throughout the rest of this paper, we will use origin "0" and all subscripts begin at "0". In APL\360 the "size of A" is written ρA, and the rank is $\rho\rho A$. The ravel of A is written $,A$. APL frequently uses one symbol for two different operations. For example, if the item to the left of "ρ" is a variable (or a bracketed expression or a function of no arguments) then ρ stands for reshape; otherwise it stands for size. The statement $C \leftarrow B\ \rho\ A$ defines an array C. The size of C is equal to B and the elements of C are the same as the elements of A. If A is a 5×6 matrix, then $C \leftarrow (3\ 5\ 2)\ \rho\ A$ sets C equal to an array of size 3 5 2 and $C[0; 0; 0] = A[0; 0]$, $C[0; 0; 1] = A[0; 1]$, $C[0; 1; 0] = A[0; 2]$, and so on. If the item to the left of a "," is a variable, then the operator stands for catenate; otherwise it stands for ravel. If A and B are vectors, then A,B is a vector consisting of the elements of A followed by the elements of B. See Falkoff and Iverson[1] for a complete description of APL.

The APL statement $Z \leftarrow A[I; J; K]$ is much more complex than the corresponding FORTRAN or PL/1 statement. The complexities of APL arise from the following causes. I, J and K need not be scalars; they may be scalars, vectors or arrays of any number of dimensions. An APL compiler or interpreter has no prior knowledge of the properties of Z; in fact Z is completely respecified by the execution of this statement. The dimensions of A, I, J and K cannot be determined until the actual execution of the statement. In a PL/1 program, the dimensions of A may vary during execution, but the dimensions are fixed once the containing procedure (or block) is entered. In APL the dimensions of A may even change during execution of the current statement; for example, $Z \leftarrow ((P,Q,R)\rho X)[I; J; K]$ is a valid APL statement which defines an array of size P,Q,R and then it immediately accesses some of the elements of this array.

A general APL subscript expression has the form $Y[I0; I1; I2 \ldots ; IK]$. Any number of subscripts are allowed, but the number of subscripts must be equal to the rank of Y. Each of the subscripts may be a constant, a variable, an expression, or it may be omitted. An omitted subscript corresponds to the PL/1 use of * in a subscript position. It implies the subscript in that position should range over all valid values. As an example, $M[3;]$ gives row 3 of matrix M. In the case of $Z \leftarrow Y[I0; I1; I2 \ldots ; IK]$ the size of Z is the concatentation of the sizes of $I0$ through IK, thus $(\rho Z) = (\rho I0), (\rho I1), \ldots, (\rho IK)$. If the Jth subscript position is empty, then $(\rho Y)[J]$ should be used in place of ρIJ. Let $\underline{I}0$ denote the ravel of $I0$, $\underline{I}1$ denote the ravel of $I1$, and so on. We will use $IKM1$ (which stands for $I\ K - 1$) to denote the penultimate subscript. The successive elements of the ravel of Z are

$$
\begin{array}{l}
Y[\underline{I}0[0]; \underline{I}1[0]; \ldots \underline{I}KM1[0]; \underline{I}K[0]] \\
Y[\underline{I}0[0]; \underline{I}1[0]; \ldots \underline{I}KM1[0]; \underline{I}K[1]] \\
\quad \vdots \\
Y[\underline{I}0[0]; \underline{I}1[0]; \ldots \underline{I}KM1[0]; \underline{I}K[NK]] \\
Y[\underline{I}0[0]; \underline{I}1[0]; \ldots \underline{I}KM1[1]; \underline{I}K[0]] \\
\quad \vdots \\
Y[\underline{I}0[N0]; \underline{I}1[N1]; \ldots \underline{I}KM1[NKM1]; \underline{I}K[NK]],
\end{array}
$$

where NK stands for the number of elements in $\underline{I}K$ minus 1. All of the subscripts used above must be valid subscripts; this implies that each element of $I0$ must be nonnegative and less than $(\rho Y)[0]$; similarly each element of $I1$ must be nonnegative and less than $(\rho Y)[1]$, and so on.

Since this definition looks complicated we will examine some cases. Let A be a three-dimensional array. If I, J and K are scalars, then $A[I; J; K]$ is a scalar and this case is similar to the FORTRAN or PL/1 case. If I and J are scalars and K is a vector with elements 5 6 2, then $A[I; J; K]$ is a vector with elements $A[I; J; 5]$, $A[I; J; 6]$ and $A[I; J; 2]$. If I is a scalar with value 4, J is a vector with elements 2 9 8 3, and K is a vector with elements 5 6 2, then $A[I; J; K]$ is a 4×3 matrix with the values

$$
\begin{array}{lll}
A[4; 2; 5] & A[4; 2; 6] & A[4; 2; 2] \\
A[4; 9; 5] & A[4; 9; 6] & A[4; 9; 2] \\
A[4; 8; 5] & A[4; 8; 6] & A[4; 8; 2] \\
A[4; 3; 5] & A[4; 3; 6] & A[4; 3; 2].
\end{array}
$$

As a final example, the following program shows how a general subscript on a three-dimensional array can be expressed in terms of simple scalar subscripts on a vector.

Program to compute $A[I; J; K]$:

```
    ∇ Z ← EVAL3 A; D; M; S; S1; S2; S3; R; I1; I2; I3
[1]     c NOTE THAT 0 ORIGIN IS USED
[2]     R ← ×/(S1 ← ρ,I), (S2 ← ρ,J), (S3 ← ρ,K)
[3]     S ← (ρI), (ρJ), ρK
[4]     Z ← Rρ1 ↑ ,A
[5]     → NULL IF R = 0
[6]     D ← ρA
[7]     M ← ¯1
[8]     I1 ← 0
[9]     DO2: I2 ← 0
[10]    DO3: I3 ← 0
[11]    LOP: Z[M ← M + 1] ← (,A)[(,K)[I3] + D[2]
              × (,J)[I2] + D[1] × (,I)[I1]]
[12]        → LOP IF S3 > I3 ← I3 + 1
[13]        → DO3 IF S2 > I2 ← I2 + 1
[14]        → DO2 IF S1 > I1 ← I1 + 1
[15]    NULL: Z ← SρZ
    ∇
```

```
    ∇ Z ← A IF B
[1]     Z ← B/A
    ∇
```

Notice that the major part of the calculation is independent of the shape of I, J and K. The program produces a vector Z which contains the appropriate values, and then in the last statement it gives the correct shape to Z.

The environment

We assume that subscript evaluation is part of an emulator or interpreter. The features of the environment that are relevant to this discussion are as follows. The user's programs are written in APL. These programs are read into the computer and they are translated from the external APL form to some suitable internal form. This process does not involve compilation; there is a one-to-one correspondence between the items of the external form and the items of the internal form. An external name such as "A" is translated into an internal name α. In languages such as FORTRAN, α would be the address of the value of A. In this implementation of APL, α is a number which is used to access the description and the value of A. During execution, the memory contains a table called the address table. The address table entry for α gives the address of a block of memory which contains data in one of the following formats:

1) $B\ D\ \alpha$
2) $B\ D\ \alpha$ data giving the value of A.
3) $B\ D\ \alpha$ data giving the value of A and ρA.
4) $B\ D\ \alpha$ data giving the value of $,A$, (ρA), $(\rho\rho A)$ and ρ,A.
5) $B\ D\ \alpha$ data giving the value of $A[0]$, $(A[1]-A[0])$, ρA.

These five forms correspond to the five cases of A having no value, A being a scalar, A being a vector, A being an array and A being a vector whose values form an arithmetic progression. The latter case is used to provide a compact representation of vectors of the form $P+Q\times\iota R$. In the formats given above, B is a count of the number of bytes in the block. D is a pattern of bits called the *descriptor*; it specifies whether A has a value, and if so whether the value is character, logical, integer or real. D also specifies whether A is a scalar, a vector, an array, or an arithmetic progression vector. The execution of the user's program is controlled by an *interpreter* (we use the word interpreter to cover both interpreters and emulators). The interpreter scans the internal form of the APL program, starting at the right hand end of the statement and moving to the left. The interpreter moves items from the APL program onto the stack until it comes to the stage when an operation can be performed. A *stack* is simply a series of successive memory locations which is used to hold a list; it is used on a first-in, last-out basis. If the interpreter starts to execute the statement $Z\leftarrow A[I;\ J+B*C;\ K+D]$ then it builds up the stack as far as

(top of stack) $\longrightarrow K+D]\ldots.$

We show the stack using external names; the real stack, however, will contain the internal representation of all names and operators. When the stack reaches the stage shown above and the interpreter evaluates $K+D$, it generates as temporary result with the name $T1$ and sets the stack to $T1]$ and continues to scan the program. In a language such as PL/1 or ALGOL it would be possible to put the value of $T1$ on the stack; in APL this is not usually feasible because $T1$ may be a large array. This array is stored in memory and its internal name is put on the stack. Eventually the interpreter will come to the stage where the stack contains $A[I;\ T3;\ T1]$, where $T3$ contains the value of $J+B*C$. At this stage, the interpreter will enter the subscript-processing routine. The subscript routine will take the top item off the stack (in this case it is the name of "A") and then using this name, it can use the address table to find the block which gives the type, size and value of A.

Although the interpreter is usually written in assembly language, it could also be written in microcode. See Breed and Lathwell[7] for an example of an interpreter and Hassitt, Lageschulte, and Lyon[8] for an example of a microcoded APL emulator. We will describe the subscript evaluation process using the APL language. It should be noted, however, that these APL programs are descriptions of machine language programs, and consequently we will take care to use only simple operations. We assume that the machine (on which the interpreter will run) can move data from one part of memory to another, that it can do additions, subtractions and multiplications, and conditional branching; we do not assume that it can do nonconventional operations such as the APL encode and decode. (We do use an encode with base 2 but this usage corresponds to a simple IBM System/360 test under mask instruction). We assume that the interpreter will have a subroutine that can allocate a block of memory and set up the appropriate address table entry and the data formats discussed earlier.

The method

No description is available in the literature of the methods used in the existing APL interpreters. We assume they use a generalization of the method shown in the *EVAL3* function. Abrams[9] describes a method for use on a theoretical APL machine, but his basic machine is so radically different from conventional machines that his techniques are not directly applicable. In the APL emulator for the IBM/360 model 25[8], we used a microcode routine for scalar and vector subscripts and an APL routine (which in turn called on the microcoded routines) to do

the general case. The method used in *EVAL3* can be modified to treat the case of a general N-dimension subscript. The routine would have to be reorganized; instead of the three explicit loops using $I1$, $I2$ and $I3$ there would be a general iteration vector $I[0], I[1], \ldots I[N-1]$ which would control N implicit loops. These changes are nontrivial but they can be done. There is however a fundamental objection to this method; the method is quite inefficient in almost all cases. The *LOOP* statement in *EVAL3* uses two multiplications, and this statement is repeated $S1 \times S2 \times S3$ times (see *EVAL3* for the definition of $S1$, $S2$ and $S3$). The method to be discussed here needs at most $S1 + S2 + S3$ multiplications. Many APL subscripts have one or more empty subscript positions (see Kolsky[2] for many examples). The method proposed here will recognize these special cases and, for example, if A is a $10 \times 20 \times 30$ array, then the method will recognize that $(,A[3;;])$ is the same as $(,A)[1800 + \iota 600]$ and it will use this fact to optimize the evaluation.

The method which we use to evaluate APL subscripts is divided into three stages. These stages are described in detail by three APL routines *INDA*, *INDB* and *INDC*. The subscript calculation begins at the stage where the stack contains information of the form

(top of stack) $\longrightarrow A[I0; I1; \ldots; IK]$,

where A is the internal name of a vector or array, and $I0$, $I1$, $\cdots$ IK are the internal names of scalars, vectors or arrays; some or all of the subscript positions may be empty. In the first stage we set up a vector called L, which gives a convenient, and usually concise description of the information contained in $I0$ through IK. In stage 2 we look for features that can be factored out of the subscript list, so as to optimize the final calculation. In stage 3 we show how the vector L can be used to access successive elements of the final answer.

• *Initialization*

We now consider the details of the first stage. Suppose the stack contains the information shown above. We build up the vector L, which contains $K + 5$ blocks of information, $L = C, B0, B1, \ldots BK, V, S, T$. C contains six integers, namely $C = 8\ K+1\ 6 \times K+1$ u u u, where u denotes unused or undefined. These undefined slots will be used at a later stage. Each of the B blocks contains six integers, with one B block corresponding to each subscript position. Each B has one of the following formats:

		$B[0$	1	2	3	4	5]
I is scalar	B is	0	I	u	u	0	u
I is AP vector	B is	1	$I[0]$	u	u	ρI	$I[1]-I[0]$
I is missing	B is	5	0	u	u	OS	1
I is vector or array	B is	2	u	u	u	ρ,I	OV

Will be used to hold $(\rho A)[J]$

Will be used to hold weights

If I is a scalar or an arithmetic progression vector, then the B block contains a complete description of I. If I is a general vector or array, then we copy the values of I into the block V and use $B[5]$ to give the offset of these values. S gives the shape information, $S \leftarrow (\rho I0), (\rho I1), \ldots (\rho IK)$ and $T \leftarrow \rho S$. If the Jth subscript is missing then a single element is used instead of (ρIJ). The offset OS will point to this element, and its value will be filled in later. Some examples should clarify this description. If the subscripted example was $A[3\ 2\ 4;\ 1\ 5\ 3\ 4;\ 2]$, then the resultant L block is

```
8  3  18  0  0   0      ← C
2  0   0  0  3  24      ← B0
2  0   0  0  4  27      ← B1
0  2   0  0  0   0      ← B2
3  2   4  1  5   3  4   ← V
3  4                    ← S
2                       ← T
```

If the subscript expression was $A[3\ 2\ 4;\ ;\ 2]$, then the L block is

```
8  3  18  0   0   0
2  0   0  0   3  24
5  0   0  0  28   1
0  2   0  0   0   0
3  2   4
3  0
2
```

If the subscript expression is $A[I0;\ 1\ 5\ 3\ 4;\ 2]$, where $I0$ is $3\ 1\ 2\ \rho\ 6\ 5\ 7\ 1\ 4\ 4$, then the L block is

```
8  3  18  0  0   0
2  0   0  0  6  24
2  0   0  0  4  30
0  2   0  0  0   0
6  5   7  1  4   4  1  5  3  4
3  1   2  4
4
```

Finally if the subscript expression is $A[2;\ \iota 9;]$, then L is

```
8  3  18  0   0  0
0  2   0  0   0  0
1  0   0  0   9  1
5  0   0  0  25  1
9  0
2
```

Appendix *I* gives the APL description of the method used to compute L. The function *INDA* will compute an L for any number of subscripts.

A possible objection to the method used in *INDA* is that we copy subscripts into L and this procedure might waste time. One advantage of the copying is that we can multiply the copy by an appropriate weight factor and store

the result in the same place; doing the multiplication at this stage effectively takes the multiplication outside the inner loop of the subscript calculation. The APL language insists that subscripts should have integer values, but there is no guarantee that they will have an integer representation. We can check for integer values and convert to an integer representation at the time the copying is done. In practice many APL subscripts are either scalars or arithmetic progression vectors, and in this case the copying takes a negligible amount of time.

• *Analysis and compression*

The next stage, which is described in *INDB*, is to check the value of the subscripts, insert the weight factors, and finally to reduce the number of subscripts, if possible. If $I0, I1, \ldots$ are scalars, then

$$A[I0; I1; \ldots IK]$$
$$= (,A)[(W[0] \times I0) + (W[1] \times I1) + \ldots W[K] \times IK],$$

where $W[K] = 1$

and $W[J] = W[J+1] \times (\rho A)[J+1]$

$$\text{for } J = K-1, K-2, \ldots 0.$$

If IJ is a vector or an array, then each element of IJ has to be multiplied by $W[J]$. Of course in the arithmetic progression case we use the fact that

$$((W[J] \times P) + (W[J] \times Q) \times \iota R)$$
$$= W[J] \times (P + Q \times \iota R).$$

As discussed earlier, L contains the blocks C, $B0$, $B1$, . . . BK, V, S, T. The routine *INDB* implements the following rules. For each block BJ, set $BJ[2] = (\rho A)[J]$ and $BJ[3] = W[J]$ and then

1) If IJ is a scalar, then check that it is in range (that is nonnegative and less than $BJ[2]$ and multiply it by the weight factor.
2) If IJ is an arithmetic progression vector, then check that $B[1]$ and $B[1] + B[5] \times (B[4] - 1)$ are in range and multiply $B[1]$ and $B[5]$ by the weight factor which is in $BJ[3]$.
3) If IJ was an empty subscript, then fill the rank information into the appropriate place in S, and now make this case look like the arithmetic progression case.
4) If IJ is a general vector or array, then check that each element is in range, and multiply each element by the weight factor. The elements are conveniently available in $L(B[5]), L(B[5]+1), \ldots$

Notice that *INDB* requires $K-1$ multiplications to compute the weight factors, but that it then needs only one multiplication for each scalar, two multiplications for each arithmetic progression vector, and n multiplications for each general array, where n is the number of elements in the array. The total number of multiplications is the sum (not the product) of the number of multiplications for each item.

If A is a $10 \times 20 \times 30$ array, and we consider the first example shown above (namely, $A[3\ 2\ 4; 1\ 5\ 3\ 4; 2]$) then the results of the above calculation would set L to

```
   8     3    18     0    0    0
   2     0    10   600    3   24
   2     0    20    30    4   27
   0     2    30     1    0    0
1800  1200  2400    30  150   90  120
   3     4
   2
```

Notice that the final result could now be obtained without any further multiplications. The successive elements of the result are taken from $A[3; 1; 2]$, $A[3; 5; 2]$, $A[3; 3; 2]$ and so on. The final routine will access these elements as $(,A)[1800 + 30 + 2]$, $(,A)[1800 + 150 + 2]$, $(,A)[1800 + 90 + 2]$, and so on.

Part of *INDB* is concerned with optimization. We can best illustrate this by considering a specific example. Suppose the subscript expression were $A[2 + \iota 3; 3 + 4 \times \iota 5; 6]$ and that A has size 10 20 30. After the *INDA* stage, L will be

```
8  3  18  0  0  0
1  2   u  u  3  1
1  3   u  u  5  4
0  6   u  u  0  u
3  5
2
```

and after the processing described above it would be

```
8     3  18    0  0    0
1  1200  10  600  3  600
1    90  20   30  5  120
0     6  30    1  0    0
3     5
2
```

The successive elements of the final result are to be taken from $A[2; 3; 6]$, $A[2; 7; 6]$, $A[2; 11; 6]$, $A[2; 15; 6]$, $A[2; 19; 6]$, $A[3; 3; 6]$, $A[3; 7; 6]$ and so on. Let R denote the ravel of A; then these elements are the same as $R[1200 + 90 + 6]$, $R[1200 + 90 + 120 + 6]$, $R[1200 + 90 + 240 + 6]$, $R[1200 + 90 + 360 + 6]$, $R[1200 + 90 + 480 + 6]$, $R[1200 + 600 + 90 + 6]$ $R[1200 + 600 + 90 + 120 + 6]$, and so on. We notice that this can be written $R[F]$, $R[F + 120]$, etc., where $F = 1200 + 90 + 6$, is the fixed part of each expression. In the routine *INDB* we calculate this constant. In the general case, if IJ is a scalar, then we can add $BJ[1]$ to F and then erase the block BJ. If IJ is an arithmetic progression vector, then we can add $BJ[1]$ to F and then set $BJ[1]$ to zero. In the final calculation (described in the next section), each B block will

control a loop in the subscript evaluation process; eliminating a B block will eliminate an unnecessary loop.

In the above example, the successive elements of the result are $R[F]$, $R[F + 120]$, $R[F + 240]$, and so on. We notice that the first five items are spaced at equal intervals, but this is not surprising since the original expression contained an $\iota 5$. We see subsequent items are also separated by 120. The separation implies that we could replace the two blocks:

```
1  __  __  __  3  600
1  __  __  __  5  120
```

by a single block of the form

```
1  __  __  __  15  120.
```

This is true because a block of the form

$BJ =$ 1 __ __ __ $BJ[4]$ $BJ[5]$

means that *INDC* should use the steps $0, BJ[5], 2 \times BJ[5], \cdots (BJ[4] - 1) \times BJ[5]$; then *INDC* should reset the counter for this block to zero and increment the counter of the prior block by its stepping factor. Let $BJM1$ (which stands for B of $J - 1$) be the prior block. If this block is also an arithmetic progression vector, then its step is $BJM1[5]$. Now if $BJM1[5] = BJ[4] \times BJ[5]$, then we can write the double loop controlled by BJ and $BJM1$ as a single loop. If both loops are used, then *INDC* will come to the stage where one counter changes from $(BJ[4] - 1) \times BJ[5]$ to zero and the other counter changes by $BJM1[5]$. The same effect can be gained by changing the first counter from $(BJ[4] - 1) \times BJ[5]$ to $BJ[4] \times BJ[5]$. The steps at statements 18 and 20 of *INDB* (see Appendix I) carry out the tests for this optimization.

We now summarize the calculations done by *INDB*. We check that each subscript is in range. Then we multiply it by an appropriate weight factor. Each B block in L will control a loop in the final calculation. We remove the constant factor from scalar and arithmetic progression vector blocks, and we erase the scalar blocks. Finally, if two arithmetic progression blocks are next to each other, we test to determine if the two blocks can be compressed into one. It is this compression which will result in the efficient calculation of special cases such as $A[K\ ;\ ;]$.

At the end of *INDB*, L has the form $C, B0, B1, \ldots V, S, T$, where

$C \leftarrow F$ u u u u u
$B \leftarrow X$ u u u N G
$V \leftarrow$ (old values $-$ index origin) $\times$ weight factors
$S \leftarrow$ shape of result with all values filled in
$T \leftarrow$ rank of result,

where if
$(4|X) = 0$ then this block is no longer used
$(4|X) = 1$ then this block generates $G \times \iota N$
$(4|X) = 2$ then this block generates $L[G + \iota N]$
F = fixed part of subscript
u = unused part of block.

The active blocks have been squeezed together. If we let X denote the first item in a block and let $BIT = (8\rho 2)\top X$, then the outermost block has $BIT[4] = 1$, the inner block has $BIT[3] = 1$.

As an example, if the subscript expression is $A[3\ ;\ ;]$ and A is a $10 \times 20 \times 30$ array, then at the end of *INDB*, L will be

```
1800   u  u  u    u  u
  29   u  u  u  600  1
unused block
unused block
  20  30
   2.
```

This result implies that $A[3\ ;\ ;] = 20\ 30\ \rho\ (,A)[1800 + \iota 600]$.

• *Evaluation*

The final stage is to show how L can be used to generate the subscripts. If the number of elements in the result is zero, then we do not need to proceed any further. If the result is a scalar, then $L[0]$ represents the complete result. If the result is nonscalar and nonnull, then we have to generate successive subscripts by progressing through $B0, B1, \cdots$. Each B has the form X u u u N G, where u is unused and X, N and G are described in the previous section. The basis of the method is to use each B to control a DO loop. We use $B[1]$ to count $N - 1, N - 2, \cdots 0$; we use $B[2]$ to hold a step or an offset, and we use $B[3]$ to hold a current value. The function *INDC* produces successive values, one at a time. We could arrange to produce all values at once, but in a typical application the rest of the interpreter will call on *INDC* for one value at a time. *INDC* does not use any global variables other than L. *INDC* does use some local variables but these are all scalars. In some compilers (or interpreters) there is a limit (typically 2, 3 or 7) on the number of subscripts, and the compiler writer can make use of this fact. He can allocate a small number of memory locations to hold intermediate results. In APL, the number of subscripts and number of elements in each subscript is unlimited; we can not assume that there is some permanent area which can be used to hold intermediate results. *INDA* assumes that the interpreter has some mechanism for allocating memory; it uses this mechanism once, to get space for L, and thereafter it saves intermediate results in L.

It will be seen that *INDC* allows a negative subscript; however, *INDB* has already checked all subscripts so this case cannot arise. The reason for allowing negative

subscripts in *INDC* will be apparent when we come to consider selection operators.

To summarize the position so far: We have presented a series of functions which describe the evaluation of APL subscripts. The method does apply to any number of subscripts, it uses a moderate amount of intermediate storage, it applies to subscripts of any size and shape, and it does analyze all cases so that the final evaluation takes advantage of all special cases.

We have assumed that the subscript is on the right of an assignment. The following steps apply to both subscripts on the left or on the right. Initially, the stack is $A[I0; \ldots$ or $A[I0] \ldots$. After the use of *INDA* and *INDB*, the stack will be

$A[L] \leftarrow B \ldots$ or $A[L]$ or $A[L]X \ldots$
where X is not the assignment operator.

The second and third cases imply subscripting on the right. The steps are:

Get space for result, space is defined by

$(-1 + {}^{-}1 \uparrow L) \uparrow L$

Use *INDC* to generate subscripts, get values and store successive results.

The first case implies subscripting on the left and the steps are:

If $(1 \neq \rho,B)$, then check that

$(\rho B) = {}^{-}1 \downarrow (-1 + {}^{-}1 \uparrow L) \uparrow L.$

If A is arithmetic and B is character (or vice versa) and if the number of subscripts specified by L is not zero, then indicate an error. If (type of A)$\neq$ (type of B), then it may be necessary to convert the whole of A. This happens if A is integer and B is real, or A is logical and B is nonlogical. Get succesive elements of B, convert to type of A, use *INDC* to find offset of result position and store result in A.

The calculations specified by *INDA*, *INDB* and *INDC* require many separate steps and it might seem that these would take an excessive amount of time. There are some saving factors. Most of the steps can be performed by a few LOAD, STORE or BRANCH instructions. A machine language MULTIPLY instruction typically takes eight times as long as a LOAD instruction. If the subscript result contains a large number of elements, then the initial analysis will be compensated by the saving in multiply times. There are several places where further optimization can be performed; for example, if the result of *INDB* shows that there is a single loop which is an arithmetic progression vector with a step of +1 or −1, then *INDC* can (except in the logical case) be replaced by a MOVE LONG instruction. There is no doubt that APL subscript operations are complicated and that any method will require many steps; because of the dynamic structure of all variables, it is not possible to do any of the analysis during a compilation stage.

Selection operators

The subscript operations of APL can be used to select part on an array; for example, $M[3\ ;]$ selects row 3 of matrix M. Other operators such as reverse and transpose can be used to select part of an array. Abrams[9] has suggested that all these operations have certain properties in common, and he applies the term *selection operation* to them. The significance of Abrams' work is that he develops an efficient and elegant way of doing selection operations.

Given an array A, Abrams uses an internal representation which contains a value part (which we will call *AV*), and an access function. The access function is divided into three parts denoted by *ABASE*, *AVEC* and *ADEL*. The definition of these quantities is:

```
        AV ← ,A
        AVEC ← ρA
        ADEL ← (N ← ρρA)ρ 1
LOOP:   → NOMOR IF 0 > N ← N − 1
        ADEL[N] ← ADEL[N + 1] × AVEC[N + 1]
        → LOOP
NOMOR:  ABASE ← 0
```

ADEL is just the weight function discussed earlier. If I is a vector, then

$A[I[0]; I[1]; \ldots] = AV[ABASE + +/ADEL \times I].$

Consider a typical selection operation such as $Z \leftarrow \phi[J]\ A$. Abrams shows that the internal representation of Z has

```
ZV ← AV
ZBASE ← ABASE + ADEL[J] × (AVEC[J] − 1)
ZVEC ← AVEC
ZDEL ← ADEL
ZDEL[J] ← −ADEL[J]
```

Abrams' implementation is arranged so that both A and Z have an access function, but they share a common value block. Abrams defines a selection operation to be any operation whose result can be computed by redefining an access function. He shows that reverse, transpose, and some cases of take, drop and subscripting are selection operations.

In the previous section we described a function *INDC* which produces successive elements of a subscripted expression. The complete expression can be computed using the function *EVAL* shown below. The APL programs shown in this section will give an over-all view; they will not show the detailed steps to be used by an interpreter.

```
    ∇ Z ← EVAL; I; J; N; T
[1]     c NOTE THAT 0 ORIGIN IS USED
[2]     → NOTNULL IF 0 ≠ J ← ×/N ← ¯1 ↓
        (−1 + ¯1 ↑ L) ↑ L
[3]     Z ← Nρ1 ↑ ,A
[4]     → 0
[5]     c THIS SETS I = ZERO OR BLANK
[6]   NOTNULL: I ← 1 ↑ (T ← 0)ρA
[7]     c SO NOW WE CAN INITIALIZE Z
[8]     Z ← Jρ I
[9]   LOOP: → SKIP IF 0 > I ← INDC T > 0
[10]    Z[T] ← (,A)[I]
[11]  SKIP: → LOOP IF J > T ← T + 1
[12]    Z ← NρZ
    ∇
```

We can now extend Abrams' idea of selection operators. The vector L used by *INDC* can be used in place of *ABASE*, *AVEC* and *ADEL*, so that L is now the access function (actually it is L in conjunction with *INDC* that carries out the access, but *INDC* does not change from case to case). An operation S is a selection operation if there is an access function L such that *EVAL* and *INDC* can compute the result of SA. We have seen that *EVAL* can be used for all cases of subscripting. It can also be used for reverse, transpose, compress, expand, take and drop. Ravel and some cases of reshape can be considered as selection operations but it is usually not profitable to do this.

As a first example, consider $\phi[J]A$. For the moment, suppose ρA is 10 20 30 and J is 0. The result is

$Z \leftarrow \phi[0]A$ or

$Z \leftarrow A[9 - \iota 10\ ;\ ;]$.

If we take $A[9 - \iota 10\ ;\ ;]$ and apply *INDA*, then it will produce the list

```
 8  3  18  0   0   0
 1  9   0  0  10  −1
 5  0   0  0  25   1
 5  0   0  0  26   1
10  0   0
 3
```

and now applying *INDB* would produce

```
5400   0   0  0    0     0
   9   0   0  0   10  −600
  17   0   0  0  600     1
  10  20  30
   3
```

and applying *INDC* to this would produce the desired result. A function to produce L in the general case is as follows:

```
    ∇ L ← J REV A
[1]     c GENERATE LIST FOR φ[J]A
[2]     c NOTE THAT 0 ORIGIN IS USED
[3]     SETUP A
[4]     N ← (ρA)[J]
[5]     L[(6 × J + 1) + 0 1 4 5] ← 1, (N − 1), N, −1
    ∇

    ∇ SETUP A; K; N; M
[1]     c SET UP A LIST FOR A[; ; . . . ; ;]
[2]     c NOTE THAT 0 ORIGIN IS USED
[3]     N ← 6 × K ← ρρA
[4]     L ← 8, K, N, 0 0 0 ,Nρ 5 0 0 0 0 1
[5]     L[10 + 6 × ιK] ← (ρL) + ιK
[6]     L ← L, (ρA), ρρA
    ∇
```

REV is a description of the overall process. In an actual implementation, the program would first calculate the size of L and then get space for L. We assume that A is not a scalar and that J has a legitimate value. The output from L has to be processed by *INDB* and then it can be used by *INDC*. It would be possible to write a *REV* which does not use *SETUP* or *INDB*. The monadic transpose operation can be expressed in terms of the dyadic transpose. The dyadic transpose does a transposition of subscripts. For example, if $Z \leftarrow 2\ 3\ 1 \backslash A$, then

$Z[I; J; K] \leftarrow A[J; K; I]$ for all I, J, K.

For $Z \leftarrow V \backslash A$ then the Nth subscript of A is the $V[N]$th subscript of Z. If some of the elements of $V[N]$ are equal to each other, then the corresponding subscripts of A become a single subscript in Z. A description for producing the subscript list of $V \backslash A$ is shown in function *TRANSPOSE*. It follows the method of Abrams[9].

```
    ∇ V TRANSPOSE A; BJ; D; S; T
[1]     c NOTE THAT 0 ORIGIN IS USED
[2]     GETDST
[3]     BJ ← SQUEEZE, 1, (0, (0, (0, S, D)))
[4]     BJ[0] ← BJ[0] + 8
[5]     BJ[¯6 + ρBJ] ← BJ[¯6 + ρBJ] + 16
[6]     L ← 0 0 0 0 0 0 ,BJ, (,S), T
    ∇

    ∇ GETDST; DELA; I; IRRA; RA; RRA; TEMP
[1]     c NOTE THAT 0 ORIGIN IS USED
[2]     IRRA ← ιRRA ← ρRA ← ρA
[3]     DELA ← ×/(1 ↓ RA, 1)[(RRA − 1)⊥IRRA
        ∘. + IRRA]
[4]     → TESTSD AFTER S ← D ← 0ρI ← ¯1
[5]   FIXSD: S ← S, ⊥/TEMP/RA
[6]     D ← D, + /TEMP/DELA
[7]   TESTSD: → FIXSD IF (∨/TEMP ← V = I ← I + 1)
```

```
[8]     S ← ((T ← ρS), 1)ρS
[9]     D ← (T, 1)ρD
    ∇

    ∇ Z ← SQUEEZE B; T
[1]     c NOTE THAT 0 ORIGIN IS USED
[2]     Z ← ¯6 ↑ B
[3]   LOOP: → 0 IF 0 = ρB ← ¯6 ↓ B
[4]     → COMBINE IF (T ← ¯6 ↑ B)[5] = Z[4]
        × Z[5]
[5]     → LOOP AFTER Z ← T, Z
[6]     COMBINE: → LOOP AFTER Z[4] ← T[4]
        × Z[4]
    ∇

    ∇ Z ← A AFTER B
[1]     Z ← A
    ∇
```

Consider $D \uparrow A$ and let all the elements of D be greater than zero. The method used for this restricted case can easily be extended to drop, and to the case where elements of D are less than or equal to zero. $D \uparrow A$ can be expressed in terms of the subscript list

$$L \leftarrow C, B0, B1, \ldots BK, S, T \quad \text{where} \quad \begin{array}{l} S \leftarrow D \\ T \leftarrow \rho,D \\ K \leftarrow T - 1 \end{array}$$

and each B usually has the form $BJ = 1\ 0\ 0\ 0\ D[J]\ 1$. This form is adequate providing $D[J] \leq (\rho A)[J]$ (Abrams' methods are restricted to this case). We can treat the case

$D[J] > (\rho A)[J]$ by

$BJ = 2\ 0\ 0\ 0\ D[J]\ OV,$

where OV is the offset of the vector

$(\iota(\rho A)[J]), (D[J] - (\rho A)[J])\rho{}^{-}1.$

The *INDB* routine will reject negative subscripts. As we mentioned earlier, however, *INDC* will accept a negative subscript. We can now give a meaning to a negative result from *INDC*. If *INDC* returns the result J, then the corresponding element of A is as follows:

if $J \geq 0$ then $(,A)[J]$
if $J < 0$ then zero or blank.

Suppose A is $2\ 2\ \rho$ *'PQRS'*, then $3\ 2 \uparrow A$ would produce the list

```
 0  0   0  0  0   0
10  0   0  0  3  18
17  0   0  0  2   1
 0  2  ¯2
 3  2
 2
```

INDC would produce $0\ 1\ 2\ 3\ {}^{-}2\ {}^{-}2$ and the result is $3\ 2\ \rho$ *'PQRS '*. An unsatisfactory feature of this method is that it requires a block of type 2 (explicit vector) rather than a block of type 1 (*AP* vector) for each $D[J] > (\rho A)[J]$. An alternative approach is to use the type 1 vector, $1\ 0\ 0\ 0\ D[J]\ 1$ in all cases, but modify *INDC* at statement 48 from

$\rightarrow LOOP, L[U + 3] \leftarrow Z \leftarrow Z + STEP$

to

$\rightarrow (L[U + 4] > L[U + 3] \leftarrow Z \leftarrow Z + STEP)/LOOP$
$\rightarrow LOOP, Z \leftarrow {}^{-}1$

and a similar modification at statement 35. An unsatisfactory feature of this second method is that it adds an extra test in the inner loop of every subscript calculation. Since the second method penalizes all cases, whereas the first method only penalizes exceptional cases, we prefer the first method.

The compress operation $Z \leftarrow V/[J]A$ is equivalent to $Z \leftarrow A[;;\ldots;C;;\ldots;]$, where there are J semicolons in front of the C, $C \leftarrow V/\iota(\rho A)[J]$ and there are ${}^{-}1 + (\rho\rho A) - J$ semicolons after the C. This result can be expressed in terms of a list L with at most three blocks. The empty subscripts reduce to a block of type 1 and the C reduces to a block of type 1 or type 2. We see that $Z \leftarrow V\backslash[J]A$ can also be treated in the same way, but C is now $C \leftarrow (- \sim V) + V\backslash\iota(\rho A)[J]$ and we rely on *INDC* to transmit a negative subscript as a request for a fill element (zero or blank).

Consider $B\phi[J]A$. If B is a scalar, then we can use the method described in the previous paragraph, where $C \leftarrow B\phi\iota(\rho A)[J]$. Alternatively, we could use $C \leftarrow B + \iota(\rho A)[J]$ and modify *INDC* so that subscripts are interpreted modulo ρA. We cannot see any way of including the general rotate operation (where B is a vector) in our existing list form. There are several methods of doing the general case, but including these methods in *INDC* would seem to penalize all the other subscript operations.

To summarize the discussion of this section: We extend Abrams' notion of a selection operation; we define a selection operation to be any operation S which can be represented by the subscript list described earlier. In other words, S is a selection operation if there exists a vector L such that *EVAL* will evaluate SA.

All cases of subscripting, take, drop, reverse, transpose, compress and expand can be expressed in this way.

Dragalong and beating

Abrams[9] has shown that many APL expressions can be evaluated more efficiently if various operations are combined and if other operations are commuted. If $S1$ and $S2$ are selection operators, then Abrams describes a machine which treats $S1\ S2\ A$ as follows: Make copy

of descriptor block of A, apply $S2$ to this block to produce a new block, apply $S1$ to the new block. Abrams calls this process *beating*. We gave an example earlier of how his machine does $\phi[J]A$ in a few simple steps. The Abrams machine delays the evaluation (*drags along*) of nonselect expressions. For example, $3 \uparrow (2\times - V)$ is evaluated as though it were written $2\times - 3 \uparrow V$ with a saving of $(\rho V) - 3$ operations. The design of Abrams machine is quite unlike any other machine, so we cannot use his methods; however, we can investigate the possibility of beating and dragalong in a more conventional machine.

We assume an APL design that is similar to the one described earlier[8]. The data descriptor can specify that an item is a scalar, a vector, an array, an *AP* vector, or a subscripted array. The *AP* vector and the subscripted array were not in the previous design. A subscripted array has the form L which results from *INDA* and *INDB* (or from *REV* or one of the other selection operator routines). We also set $L[1]$ equal to the internal name of the operand (in the earlier part of this paper, the operand was always called A). In the following description we use L, $L1$, $L2$, . . . to denote subscripted arrays.

In the previous design, the expression $A[I; J] + B[P; Q]$ was evaluated as follows: Form temporary result $T1 \leftarrow B[P; Q]$, form temporary result $T2 \leftarrow A[I; J]$, form $T1 + T2$. In the new design we can evaluate it as follows: Form $L1$ from $B[P; Q]$, form $L2$ from $A[I; J]$, access the elements described by $L1$ and $L2$ and add them element by element. Since the intermediate results $T1$ and $T2$ are not formed, there will usually be some saving. This saving has to be balanced against the added complexity of the design. We have, for example, to deal with expressions such as $(B \leftarrow X)[I; J] + B[P; Q]$, where $L1$ refers to a B which is no longer the current B.

There seems to be no theoretical difficulty in beating of selection operators, although in practice the algorithms are complex. The reason for the complexity is that our L lists are more general than the Abrams array descriptors. Consider the following examples:

```
A ← 10 20 30 ρ ι 6000
B ← (1 0 1 0)/[1] A[0 3 1; 2 4 7 9; 8 2].
```

The evaluation of the $A[\ldots]$ will produce the subscripted array

```
L ←  0    α    0    0    0    0
    10    0    0    0    3   24
     2    0    0    0    4   27
    18    0    0    0    2   31
     0 1800  600   60  120  210  270  8  2
     3    4    2
     3
```

where α denotes the internal name of A. The shape of $A[\ldots]$ is given by $L[33\ 34\ 35]$ and the rank is in $L[36]$. After the compression, the shape along dimension number one (dimensions are numbered 0, 1, 2) is 2 so we need $L[34] \leftarrow 2$. The block $L[12 + \iota 6]$ and $L[27 + \iota 4]$ describes the subscripts along the second dimension; now

```
60 210 = ((1 0 1 0)/60 120 210 270)
       = (1 0 1 0)/L[27 + ι4],
```

so the final result is

B = subscripted array described by

```
 0    α    0    0    0    0
10    0    0    0    3   24
 2    0    0    0    2   27
18    0    0    0    2   31
 0 1800  600   60  210    u    u    8    2
 3    2    2
 3
```

where u denotes "undefined" or "unused" and we have underscored the elements that have changed[10]. This case was quite straightforward; we now illustrate the two complications that can arise. Consider

```
A ← 10 20 30 ρ ι6000
B ← (1 0 1 1)/[1] A[0 3 1; 2 4 6 8; 8 2].
```

The resulting L is almost the same, except that $L[12 + \iota 6]$ is now 1 0 0 0 4 120, that is, it describes an *AP* vector. (1 0 1 1) applied to the *AP* vector produces a non-*AP* vector so we have to convert to a block of type 2. The second complication arises in cases such as

```
A ← 20 10 ρ ι200
C ← 3 4 ρ 0 3 1 2 4 7 9 8 2 5 1 6
B ← (1 0 1 0)/[1] A[C; 8 2].
```

After the $A[\ldots]$, the subscripted array is

```
L ←  0    α    0    0    0    0
    10    0    0    0   12   18
    18    0    0    0    2   30
     0   30   10   20   40   70   90   80   20   50   10   60   8   2
     3    4    2
     3
```

The shape of the result is still 3 4 2 so the compression is legal, but now one block (namely, $L[6 + \iota 6]$) covers two of the dimensions. It is possible to unscramble the dimensions. Suppose an L contains K blocks (in this case $K = 2$) and let $S \leftarrow {}^{-}1 \downarrow (-1 + {}^{-}1 \uparrow L) \uparrow L$ (in this case $S \leftarrow 3\ 4\ 2$). Then $L[10]$, $L[16]$, . . . $[10 + 6 \times K - 1]$ contains the number of elements covered by each block. Now $L[10]$ equals $S[0]$ or $S[0] \times S[1]$ or $S[0] \times S[1] \times S[2]$ or . . . (in this case, $L[10] = S[0] \times S[1]$) and in general if $L[10] \leftarrow \times/S[0,1,2\ \ldots\ I0]$ then $L[16] \leftarrow \times/S[(I0 + 1), (I0 + 2), \ldots I1]$ and so on. Hence, the first block came from an array of dimensions $S[0, 1 \ldots 10]$, the second block came from an array of dimensions

$S[I0 + 1, \ldots I1]$, and so on. In our example, the final result is that $L[6 + \iota 6]$ has to describe the block

```
(1 0 1 0)/3 4 ρ  0 30 10 20
                40 70 90 80
                20 50 10 60
```

that is,

```
3 2 ρ  0 10
       40 90
       20 10
```

so the final value of L is

```
 0  α  0  0  0  0
10  0  0  0  6 18
18  0  0  0  2 30
 0 10 40 90 20 10 u . . . u 8 2
 3  2  2
 3
```

If SA (where S is some selection operator) produces a subscripted array L, then L contains all the information on the value and shape of SA, hence it is always possible to compute $S'SA$ (where S' is another selection operator) by operating on L.

A final point about subscripted arrays is that if A is of length 4, then $A[0\ 2]$ and $(1\ 0\ 1\ 0)/A$ produce the same list and, therefore $((1\ 0\ 1\ 0)/A) \leftarrow X$ should have the same effect as $A[0\ 2] \leftarrow X$.

To summarize this discussion: If we defer the evaluation of subscripted arrays, then we can do beating on selection operators, we can allow selection in assignments, we may achieve increased efficiency in the evaluation of scalar operations on subscripted arrays. We do not see a simple method of implementing the general type of dragalong that is used in the Abrams machine.

Conclusions

We have described a method of calculating array subscripts of arrays. This method is completely general. It requires only a moderate amount of intermediate memory. It does take advantage of the special properties of particular subscript patterns. Wherever possible, it extracts scalar subscripts, and collapses several of the implied DO loops into a single loop. The techniques used in the subscripting case are extended to the other selection operators; namely, take, drop, reverse, transpose, compress and expand. It is shown that sequences of selection operators could be collapsed into a single selection operation.

Appendix I

The following APL functions give a detailed description of the methods described in the text. *INDA* assumes that the global variable *STACK* contains a representation of the stack, and that there is a function called *POP* which will take the top item off the stack. $STACK[SP]$ is the top item. If *POP* finds a variable name, then it sets X equal to the value of the variable, and it removes two items from the stack; the two items are the name and the following ';' or ']'. *POP* also returns a value which is 0, 1 or 2 depending on whether the variable is a scalar, an arithmetic progression vector, or a general vector or array. If *POP* finds an empty subscript position, then it returns the value 5, and takes one item off the stack. *INDA* makes two passes over the stack. In the first pass it computes the space needed for L. It gets the space for L; in a real implementation, statement 13 of *INDA* would be a call to the space management routine. In the second pass, *INDA* computes L. The global variable *IORG* in *INDB* is the APL index origin.

```
     ∇ INDA; LL; ARANK; SRANK; SP; BB; BS; BV;
       N; X
[1]    c NOTE THAT 0 ORIGIN IS USED
[2]    c COMPUTE SPACE NEEDED FOR L
[3]    LL ← 7 + ARANK ← SRANK ← 0
[4]    SP ← 2
[5]  L1: ARANK ← ARANK + 1
[6]    → (T0, T1, T2, 0 0 ,T5)[POP]
[7]  T5: → T1, SP ← SP − 1
[8]  T2: LL ← LL + ρ,X
[9]    → T0, SRANK ← SRANK + ρρX
[10] T1: SRANK ← SRANK + 1
[11] T0: LL ← LL + 6
[12]   → L1 IF(ρSTACK) > SP ← SP + 2
[13]   c GET SPACE FOR L
[14]   L ← (LL + SRANK)ρ 0
[15]   L[0 1 2] ← 8, ARANK, 6 × ARANK
[16]   L[¯1 + ρL] ← SRANK
[17]   BB ← 6
[18]   BV ← L[2] + 6
[19]   BS ← (ρL) + ¯1 − SRANK
[20]   c FORM L = C, B, V, S, T
[21]   c BB, BV, BS POINT TO NEXT PART OF B, V, S
[22]   SP ← 2
[23] L4: → (TT0, TT1, TT2, 0 0 ,TT5)[POP]
[24] TT5: L[BB + ι6] ← 5 0 0 0 ,BS, 1
[25]   SP ← SP − 1
[26] L2: → L3, BS ← BS + 1
[27] TT2: 'DOMAIN' ERRORIF ∨/,X ≠ ⌊X
[28]   L[BV + ιN ← ρ,X] ← ,X
[29]   L[BS + ιρρX] ← ρX
[30]   BS ← BS + ρρX
[31]   L[BB + ι6] ← 2 0 0 0 ,N, BV
[32]   → L3, BV ← BV + N
[33] TT1: L[BS] ← ρX
[34]   → L2, L[BB + ι6] ← 1, X[0], 0 0 ,(ρX), X[1]
       − X[0]
[35] TT0: 'DOMAIN' ERRORIF X ≠ ⌊X
```

```
[36]     L[BB + ⍳6] ← 0, X, 0 0 0 0
[37]   L3: BB ← BB + 6
[38]     → L4 IF(ρSTACK) > SP ← SP + 2
      ∇

      ∇ A ERRORIF B
[1]      → 0 IF ~ ∨/,B
[2]      A,'  ERROR'
[3]      →
      ∇

      ∇ INDB; D; W; S; K; F; Q
[1]      ⍝ NOTE THAT 0 ORIGIN IS USED
[2]      'RANK' ERRORIF L[1] ≠ ρρA
[3]      D ← (ρA)[Q ← ¯1 + ρρA]
[4]      S ← W ← 1 + F ← 0
[5]      L[K] ← L[K ← L[2]] + 16
[6]      ⍝ START AT INNER BLOCK AND PROCESS
[7]      ⍝ ONE BLOCK AT A TIME
[8]    LOOP: L[K + 2 3] ← D, W
[9]      → (T0, T1, T2, 0 0 ,T5)[8|L[K]]
[10]   T0: F ← F + W × CHECK L[K + 1] − IORG
[11]   MOR: S ← S × L[K + 4]
[12]   MOR2: → ENDLOOP IF L[K ← K − ⍳] > 7
[13]     W ← W × D
[14]     → LOOP, D ← (ρA)[Q ← Q − 1]
[15]   T5: → L1, L[K + 4] ← L[L[K + 4]] ← D
[16]   T1: F ← F + W × CHECK L[K + 1]
[17]     → (0 × CHECK L[K + 1] + L[K + 5]
         × L[K + 4] − 1)/L1
[18]   L1: L[K + 5] ← W × L[K + 5]
[19]     → MOR IF(L[K] > 16) ∨ 1 ≠ 2|L[K + 6]
[20]     ⍝ SUCCESIVE AP VECTORS, REDUCE IF
         POSSIBLE
[21]     → MOR IF L[K + 5] ≠ L[K + 10] × L[K + 11]
[22]     L[K + 6] ← L[K + 6] − 4|L[K + 6]
[23]     S ← S × L[K + 4]
[24]     → MOR2, L[K + 5 4] ← L[K + 11],
         L[K + 4] × L[K + 10]
[25]   T2: L[P] ← W × CHECK L[P ← L[K + 5]
       + ⍳L[K + 4]] − IORG
[26]     → MOR
[27]     ⍝ NOW MOVE ACTIVE BLOCKS TO HEAD
           OF LIST
[28]   ENDLOOP: P ← 6 + K ← 0
[29]   L4: → L2 IF 0 = 4|L[K ← K + 6]
[30]     → L3 IF K = P
[31]     L[P + ⍳6] ← L[K + ⍳6]
[32]   L3: P ← P + 6
[33]   L2: → L4 IF ~ L[K] > 15
[34]     L[6] ← 8 + 8|L[6]
[35]     L[P − 6] ← 16 + 16|L[P − 6]
[36]     L[0] ← F
      ∇

      ∇ Z ← CHECK A
[1]      → (∨/0 > ,Z ← A)/ERROR
[2]      → (~ ∨/D ≤ ,A)/0
[3]    ERROR: 'INDEX ERROR'
[4]      →
      ∇

      ∇ Z ← INDC  T; U; S; COUNT; STEP; W
[1]      ⍝ NOTE THAT 0 ORIGIN IS USED
[2]      → FETCH IF T
[3]      ⍝ INITIALISE LOOPS AND GET FIRST VALUE
[4]      Z ← L[0]
[5]      L[1] ← U ← 0
[6]    LOOP: S ← 256|L[U ← U + 6]
[7]    LOOP1: STEP ← L[U + 5]
[8]      COUNT ← L[U + 4] − 1
[9]      → (DOWN00, DOWN01, DOWN10)
         [(2 × BIT 6) + BIT 7]
[10]     ⍝ CAN ONLY OCCUR IF 0 = ρρA
[11]   DOWN00: → 0
[12]     ⍝ AP CASE
[13]   DOWN01: → (0 ≤ L[U + 3] ← Z)/POS
[14]     STEP ← 0
[15]   POS: → (BIT 3)/INNER
[16]   NOTIN: → LOOP, L[U + 1 2] ← COUNT, STEP
[17]   INNER: U ← 1 + 4 × U
[18]     L[4] ← Z
[19]   INNER1: → 0, L[1 2 3] ← U, COUNT, STEP
[20]     ⍝ GENERAL VECTOR OR ARRAY
[21]     ⍝ 'STEP' ACTUALLY CONTAINS OFFSET
[22]   DOWN10: W ← Z
[23]     Z ← (¯1, Z + L[STEP])[L[STEP] > 0]
[24]     → INNER2 IF BIT 3
[25]     → NOTIN, L[U + 3] ← W
[26]   INNER2: U ← 3 + 4 × U
[27]     → INNER1, L[4] ← W
[28]     ⍝
[29]     ⍝ GET NEXT VALUE, L[2 3 4] CONTAINS
[30]     ⍝ COUNT, STEP OR OFFSET, OLD VALUE
[31]   FETCH: → NONSC IF 2|S ← 256|L[1]
[32]   FINAL: 'NO MORE ELEMENTS IN LIST'
[33]     → Z ← 0
[34]   NONSC: → ENDS IF 0 > L[2] ← L[2] − 1
[35]     → VEC IF BIT 6
[36]     → 0, Z ← L[4] ← L[3] + L[4]
[37]   VEC: → 0 IF 0 > Z ← L[3]
[38]     → 0, Z ← L[4] + L[L[3] ← L[3] + 1]
[39]     ⍝ INNER LOOP IS FINISHED, TRY NEXT
           OUTER
[40]   ENDS: S ← 256|L[U ← ⌊U ÷ 4]
[41]   UP: → FINAL IF BIT 4
[42]     S ← L[U ← U − 6]
[43]     Z ← L[U + 3]
[44]     STEP ← L[U + 2]
```

```
[45]     → UP IF 0 > L[U + 1] ← COUNT
         ← L [U + 1] − 1
[46]     c OUTER IS OK, SO RE-CYCLE INNER
[47]     → DOWN1 IF BIT 6
[48]     → LOOP, L[U + 3] ← Z ← Z + STEP
[49]   DOWN1: STEP ← L[U + 2] ← L[U + 5]
[50]     → LOOP, Z ← (¯1, Z + L[STEP]) [L[STEP]
         > 0]
    ∇
```

```
   ∇ Z ← BIT K
[1]    c GET BIT K OF S
[2]    c NOTE THAT 0 ORIGIN IS USED
[3]    Z ← (,(8ρ2)⊤S)[K]
   ∇
```

References and notes

1. A. D. Falkoff and K. E. Iverson, *APL\360 User's Manual*, IBM Research Center, Yorktown Heights, New York (1968). Now available as IBM manual, file number H20-0683.
2. H. G. Kolsky, "Problem formulation using APL," *IBM Systems Journal*, **8** No. 3, 204–219 (1969).
3. F. E. Allen, "Program Optimization," *Annual Review of Automatic Programming*, edited by M. I. Halpern and C. J. Shaw, Vol. 5, Pergamon Press, 1969, 239–307.
4. F. R. A. Hopgood, *Compiling Techniques*, American Elsevier, New York, 1969.
5. V. A. Busam, and D. E. Englund, "Optimization of Expressions in FORTRAN," *Comm. ACM*, **12**, 666–674 (1969).
6. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Comm. ACM*, **12**, 13–30 (1969).
7. L. M. Breed and R. H. Lathwell, *The implementation of APL\360*. Interactive Systems for Applied Mathematics. Academic Press, New York. (1968), pages 390–399.
8. A. Hassitt, J. W. Lageschulte and L. E. Lyon, "Implementation of a High Level Language Machine". Preprints, ACM 4th Annual Workshop on Microprogramming, September 13–14, 1971.
9. P. S. Abrams, "An APL Machine," Stanford University Computer Science Dept. STAN-CS-70-158 (1970). Also available from U.S. Dept. of Commerce, National Technical Information Service, Document AD 706 741.
10. By coincidence, the new value of L[27] is the same as the old value.

Received June 28, 1971

The authors are located at the IBM Scientific Center, 2670 Hanover Street, Palo Alto, California 94304.