TRANSLATION OF APL TO OTHER HIGH-LEVEL LANGUAGES

Margaret M. Jacobs

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

1975

The code generated corresponding to a particular APL routine will not at first be very efficient. However, methods of optimising the generated code are discussed at length in the thesis. A brief comparison is made with other possible methods of conversion.

There are certain restrictions on the types of APL statements able to be handled by the translation method. These restrictions are listed in an accompanying appendix.

Throughout the text, several examples are given of the code which will be generated from particular APL statements or expressions. Some more lengthy examples of conversion of APL routines to FORTRAN are provided as an appendix.

TRANSLATION OF APL TO OTHER HIGH-LEVEL LANGUAGES

MARGARET M. JACOBS

ABSTRACT

The research work required to produce this thesis was carried out in the Department of Computational Science, University of St. Andrews. Financial assistance was provided by the Science Research Council.

The thesis describes a method of translating the computer language APL to other high-level languages. Particular reference is made to FORTRAN, a language widely available to computer users. Although gaining in popularity, APL is not at present so readily available, and the main aim of the translation process is to enable the more desirable features of APL to be at the disposal of a far greater number of users. The translation process should also speed up the running of routines, since compilation in general leads to greater efficiency than interpretive techniques. Some inefficiencies of the APL language have been removed by the translation process. The above reasons for translating APL to other high-level languages are discussed in the introduction to the thesis.

A description of the method of translation forms the main part of the thesis. The APL input code is first lexically scanned, a process whereby the subsequent phases are greatly simplified. An intermediate code form is produced in which bracketing is used to group operators and operands together, and to assign priorities to operators such that sub-expressions will be handled in the correct order. By scanning the intermediate code form, information is stacked until required later. The information is used to make possible a process of macro expansion. Each of the above processes is discussed in the main text of the thesis. The format of all information which can or must be supplied at translation time is clearly outlined in the text.

To

MY HUSBAND AND PARENTS

A description of a method of translating APL into other high-level languages, with particular reference to FORTRAN.

The work for this project was carried out in the Department of Computational Science, University of St. Andrews, for the degree of Ph.D. The project commenced on the 10th October, 1971.

The research for the subject matter of this thesis has been carried out by myself, and the thesis has been composed by myself. The thesis has not been accepted in fulfilment of the requirements of any other degree or professional qualification.

Margaret M. Jacobs

ACKNOWLEDGEMENTS

I would like to thank Professor A.J. Cole for his excellent supervision of my work throughout the course of my studies in St. Andrews and for his advice on the preparation of my thesis.

I am extremely grateful to all the members of staff of the John Honey Building at the University of St. Andrews for their willing assistance and encouragement, and to my fellow research students for their helpful comments.

I also must thank Mrs. V. Butterworth for typing the text of my thesis, and the Science Research Council for their financial assistance.

CONTENTS

INTRODUCTION

The following text describes a method of translation of APL to other high-level languages. The version of APL able to be translated is that described in the IBM APL 360-OS and APL 360-DOS User's Manual, with a few restrictions. These restrictions are listed in Appendix 6.

Throughout the text, the target language is assumed to be FORTRAN, but similar techniques can be applied to translate from APL to ALGOL or to PL/1. In generating the target language code, only a subset of the permissible FORTRAN statements has been used. The subset was chosen such that its members (as far as possible) have counterparts in ALGOL and PL/1. This facilitates conversion to either of these languages instead of FORTRAN. The ease of conversion to ALGOL or to PL/1 is discussed in Appendix 7.

The translation was intended in the first place to handle conversion of APL subroutines and functions, but main programs may also be translated. The APL routines are not intended to remain interactive after conversion, but to be run under a batch-processing system.

There have previously been some attempts to produce a batch-processor for APL. One such attempt was made by H. Van Hedel, who implemented an APL batch-processor in PL/1 for the IBM/360. The only restriction he imposed was that function names and local variable names should be distinct. (This restriction, among others, has been placed on the types of APL statements able to be converted to FORTRAN.) In Van Hedel[7] the following example is quoted:

```
∇R ← F1 X
  R ← ⌹X∇
∇G1 ; F1
  H
  ∇

∇G2
  H ∇
∇H
  Z ← F1-A
  ∇
```

Such an example creates ambiguity in the source text, for H returns different values in G1 and G2 .

For an interactive interpreter it is important that each operation is executed as soon as sufficient information is gathered. For a batch-processor, as much as possible of the analysis has to be done before the execution takes place.

The above example is ambiguous to a compiler, but not to an interpreter.

It is intended that only working routines be converted to other languages. Thus, the amount of checking required during conversion is greatly reduced. It can be assumed, for example, that all dimensions are conformable in matrix operations.

The reasons behind the translation (see Sayers[5]) are as follows:

(i) It is intended to provide a more easily transportable system. There are at present more FORTRAN compilers than APL interpreters. Since APL is highly suited to the development of algorithms (Smillie[6]), it would be very convenient to be able to use these algorithms on a larger scale. This would be possible if the APL routines were translated to FORTRAN. (The same argument can be applied to the translation of APL to ALGOL or PL/1.)

To make transport of the converted routines as convenient as possible, the user is provided with an option to specify the output medium for the converted routines.

(ii) A secondary aim was to improve run-time efficiency by using compilation rather than interpretation. The amount of code to be interpreted is reduced if the user supplies some information about non-scalar variables. The more information supplied, the greater the amount of compilation possible. In many cases, the types and dimensions of variables will not change, and such examples readily lend themselves to the improvement of run-time efficiency. The method of supplying extra information to improve run-time efficiency is described in Chapter I.

(iii) It is hoped that code can be optimised during the course of translation by the removal of some of the inefficiencies of APL. An example of an inefficient APL expression is

```
4 ↑ A+B
```

where A and B are non-scalar. This is obviously inefficient as all the elements of A and B are summed, whereas only four summations are essential. The method of removing the above inefficiency is outlined in the following text.

In December, 1971, V.L.Moruzzi gave a set of simple rules for translating from APL to FORTRAN by hand. He estimated that mechanical APL/FORTRAN translations could achieve a 3Ø-fold reduction in CPU time. This is discussed in Moruzzi[3].

At a private meeting, Dr. J.L.Alty of Liverpool University remarked that, after visiting various APL installations in the U.S.A. and Canada, he found APL three to four times faster than other languages for program development, but one hundred times slower for execution. This result, he stated, emphasized the need for interchangeability between APL and other languages.

The translation from APL to FORTRAN is effected by a series of macro expansions. The order of expansion of macros is determined by the order of the operators in the APL source test.

A system of bracketing was introduced to ensure that all operators (and hence macro expansions) would be assigned the correct priorities. Reverse polish techniques could also have been applied during the translation process. The rival merits of each method are discussed in Appendix 5.

The options available to the user are discussed in Chapter I, together with the method of storage allocation. A lexical scan of the APL source text is first carried out to simplify the subsequent processes. The lexical scanning phase is discussed in Chapter II. Brackets are then introduced during a right-to-left scan of the code and an intermediate code form is set up. This is discussed in Chapter III. Stacking of information to be used as parameters for macros is described in Chapter IV, while the macro method itself is dealt with in Chapter V. A discussion of labels and jumps is given in Chapter VI, while Chapter VII describes the pre-optimisation phase. A process whereby the generated code can be optimised has been devised. It is described in Chapter VIII.

The APL-FORTRAN translator is written entirely in FORTRAN.

Definitions of the names used in the following text are given in Appendix 8.

CHAPTER I

INPUT PHASE AND METHOD OF STORAGE ALLOCATION

APL routines to be converted to FORTRAN are read, line by line, into a character array LINE. The routines are preceded by some additional information. Some of the information provided is essential to the conversion method (see §1.3) , and some can be provided as a user option (see §1.4) .

Most of the additional information supplied relates to the use of non-scalar variables. The conversion routines use a fairly complicated method of storage allocation for non-scalar variables. The method is necessarily complicated as the dynamic storage capability of an APL interpreter has to be simulated. The storage allocation method is discussed in detail in §1.2 . The subsequent accessing of non-scalar variables is necessarily time-consuming, as interpretive techniques have to be employed. However, under certain circumstances, a simpler storage allocation method can be employed, which reduces the access time for non-scalars considerably. The simpler method is only possible if the user supplies additional information about his non-scalar variables.

A set of APL routines can be converted to FORTRAN during one run of the conversion program. A calling program can be supplied with a set of subroutines or functions, but this has limited use in practice as the user of the converted routines may want results for several different parameter sets. Only one set of information is supplied initially for non-scalars, and this requires some care. Consider the following set of subroutines:

```
∇ A  FN  B ; X
      ___
      ___
      ___
       ∇

∇ F   Y ; X
      ___
      ___
      ___
       ∇
```

If FN and F are to be translated during a single run of the conversion routines, then the types of X in both subroutines must be the same. That is, if X is declared to be non-scalar at the start, it will be assumed non-scalar in each subroutine. There is no serious restriction when X is required to be scalar in one subroutine and non-scalar in another. The problem is easily solved by changing the variable name X in one or the other of the two routines. No problem would arise if X was a global variable, as its type would be the same in both FN and F .

Additional information must be supplied for both literal and numeric non-scalars. All information supplied is printed out. It should be noted that declarations should be supplied only for those variables which are non-scalar at their first occurrence. Otherwise, scalar occurrences of the variables would not be recognised as such.

1.1 Input of the APL Source Program

The source program is assumed initially to be in APL internal Z-code form. The program is read in and converted line by line. Each line is stored in turn in the character array LINE, which is accessed during the lexical scanning phase described in Chapter II.

During testing of the conversion routines, it was found simpler to use an input form more suited to the character set of the IBM ∅29 card punches. As far as possible, APL symbols were represented by their counterparts on the keyboard of the IBM ∅29 card punch. Composite symbols were used to represent the extraneous APL symbols. The actual representation of the APL character set used during testing is shown in Appendix 1. The symbols were then converted as required to APL internal Z-code form.

Under normal running conditions, the input would, of course, be in APL internal Z-code form.

1.2 Method of Storage Allocation

The amount of storage space allocated for an APL non-scalar variable can vary dynamically. The facility of dynamic storage allocation is not available in FORTRAN. For this reason, it was necessary to simulate the feature in the FORTRAN code produced. An arbitrary amount of storage space (represented by array YSTORE) was thus set aside for storage of all non-scalar variables, and storage space is allocated as required for individual non-scalar variables.

Since storage is allocated dynamically, a method had to be devised of linking together the various blocks of YSTORE associated with a particular non-scalar. It was obviously not advisable to link together individual locations, as the cost in terms of storage space and access time would have been prohibitive. Thus the array YSTORE was treated as separate units of 1∅ locations each. The number 1∅ was chosen as an experiment, but can be altered if found to restrict the efficiency of the resultant FORTRAN code. In practice, this means that a vector of (n*1∅+1) elements, where ∅ ≤ n will have (n+1)*1∅ locations allocated for it. A compromise had to be reached between the allocation of unnecessary locations for non-scalars and the number of linkage elements required for particular block sizes.

The information required for linking the various blocks of YSTORE is held in a separate array ZSTORE. This array also incorporates a free space list. Storage is not actually allocated for non-scalars during conversion, but the appropriate subroutine calls are generated so that dynamic allocation can take place as required during run-time of the converted routines.

To allocate or de-allocate storage for a non-scalar variable it is only necessary to update entries in the dope vector table DOPES, the array ZSTORE, and the array ZBONDS, which containslimit information for the dimensions of each non-scalar.

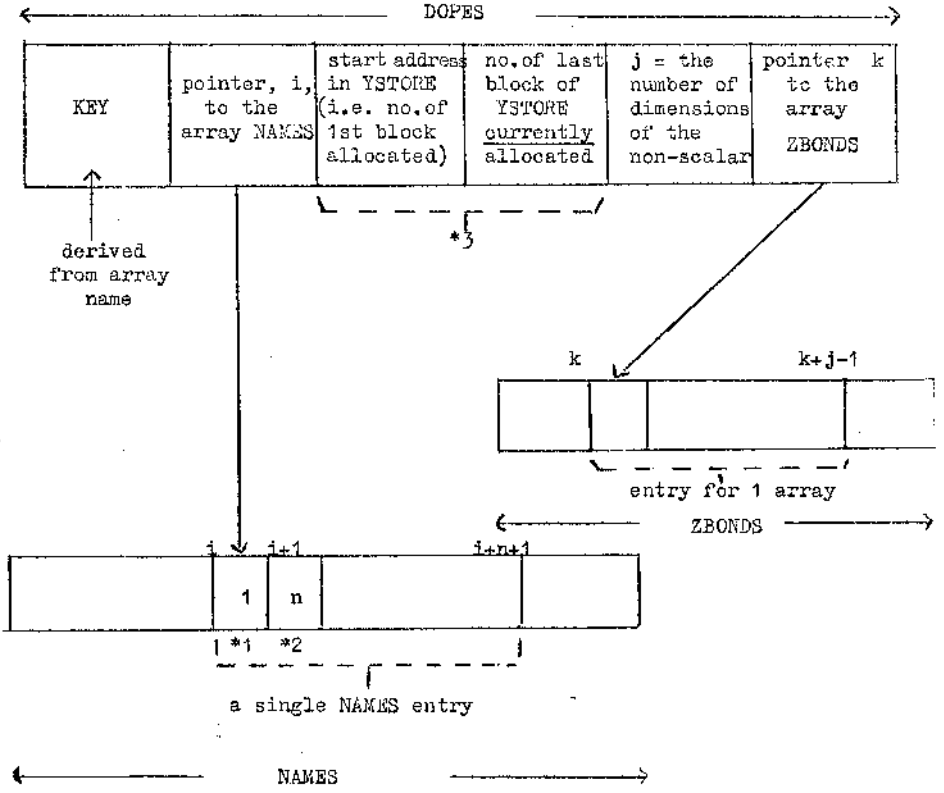
The functions of these 3 arrays are now discussed in greater detail.

1.2.1 The dope vector table DOPES

Corresponding to each non-scalar variable name in an APL routine, a 6-part entry is set up in the array DOPES. The typical form of a dope vector entry is shown in Diagram 1.2(a) .

For literal non-scalars no space is set aside in YSTORE, and the format of the dope vector entry is simplified. The third and fourth parts are not required. This is again referred to in Chapter II .

The dope vector entries may change during a subsequent optimisation phase. This phase will be undergone by the output code if the user supplies additional information about his non-scalar variables. These changes, connected with simplification of the storage and accessing mechanisms, are discussed in § 1.4 .


DOPES
KEY
pointer, i, to the array NAMES
start address in YSTORE (i.e. no.of 1st block allocated)
no.of last block of YSTORE currently allocated
j = the number of dimensions of the non-scalar
pointer k to the array ZBONDS
*3
derived from array name
k
k+j-1
entry for 1 array
ZBONDS
i
i+1
i+n+1
1
n
*1
*2
a single NAMES entry
NAMES

*1 type indicator for a non-scalar (numeric) variable is 1 (see Chapter II)

*2 n = the number of characters in the non-scalar variable name

*3 these entries will initially be the same

Diagram 1.2(a) : A typical dope vector table entry.

The 6 columns of a dope vector entry hold the following information:

(i) A key derived from the non-scalar variable name.

Only the first 3 characters of a non-scalar variable name are used to determine the key (or the first n characters, if the name has n <3 characters). The average of the Z-code values of the characters is found, and a constant subtracted such that the lowest possible key will have value 1.

The key determines the first address in DOPES to be searched when an entry is added to the dope vector table, or an existing entry is accessed.

(ii) A pointer to the array NAMES, which holds information relating to identifier names (see Chapter II).

(iii) The number of the ZSTORE element associated with the first block of YSTORE assigned to the non-scalar variable.

(iv) The number of the ZSTORE element associated with the last block of YSTORE currently allocated for the non-scalar variable.

(v) The number of dimensions of the non-scalar variable.

(vi) A pointer to the array ZBONDS where information relating to the current upper bounds of the non-scalar is stored.

An "open hash" technique is used to place entries in the dope vector table. The first address to be accessed in DOPES is given by the "key" value obtained. Hence the necessity for the lowest possible key to have value 1. If this address is empty, the location is free and is used to store the dope vector entry. Otherwise a new address is calculated and tested, and the process is repeated until a free location is found. This location is then used to store the dope vector entry.

The method of subsequent address calculation is outlined below. For a dope vector table with n rows, the address is increased by an integer m each time. If the address, j, to be searched becomes greater than n, then j is set to j - n and the process repeated until all locations of the table have been accessed. The integers n and m should be coprime to ensure that all positions of the dope vector table will be accessed. In practice, n is 64 and m is 3, this value being preferable to 1 in order to avoid the clustering of entries which might otherwise result. The open hash technique is described in Hopgood[1].

A similar method is used to access previously stored entries in DOPES. However, in this instance the test is for an entry with a key value equal to that derived from the non-scalar name. If such an entry is found it is not necessarily the required dope vector entry, since keys are not necessarily unique. (For example, A and AA or B and ABC will have identical key values.) For this reason both the key and a pointer to the array NAMES must be contained in each dope vector entry. When keys match, the characters of the non-scalar variable name and those stored in the appropriate NAMES entry must be compared to ensure that the correct dope vector entry has been found.

1.2.2 The array ZSTORE

Elements of ZSTORE can have one of two forms, depending on whether the associated block of YSTORE is a unit in the free space list or a block allocated for a particular non-scalar.

The association between ZSTORE elements and YSTORE blocks is such that ZSTORE (i) refers to the block YSTORE (1 + (i-1)*1ø) to YSTORE (i*1ø), where $1 \leq i \leq n$, n being the total number of blocks of YSTORE.

For an unallocated block, i, of YSTORE, the associated ZSTORE element has value j, where j is

EITHER (a) the number of the next block of YSTORE on the free space list,

OR (b) ∅ if i is the number of the last block of YSTORE on the free space list.

The form of a ZSTORE element associated with an allocated block of YSTORE is shown in Diagram 1.2(b) . The usage of the array ZSTORE is discussed later.

The method can be extended to cover the case where ZSTORE has more than 255 elements, that is, there are more than 255 blocks of YSTORE. There is room for expansion due to the unused 8 bits at the left-hand side of each entry.

1.2.3 The array ZBONDS

ZBONDS contains the current bounds for each non-scalar variable (literal and numeric) appearing in an APL routine. It can be updated dynamically, as can DOPES and ZSTORE.

The sixth column of a dope vector entry defines the start of bound information for the corresponding non-scalar. The number of locations of ZBONDS assigned to a particular non-scalar is obtainable from the fifth column of its dope vector entry.

In addition, a pointer ZBPTR is maintained, which gives the first free location of ZBONDS at any stage. This is useful if a new entry has to be added to ZBONDS.

The ZBONDS entry for an n-dimensional non-scalar with upper bounds $b_1, b_2, \ldots, b_n$ is shown in Diagram 1.2(c) .

← ZSTORE (m) →

Ø	N	I	J

N is the number of elements in the m^{th} block of YSTORE

I is either

(a) a backward pointer to the previous block of YSTORE allocated for the same array

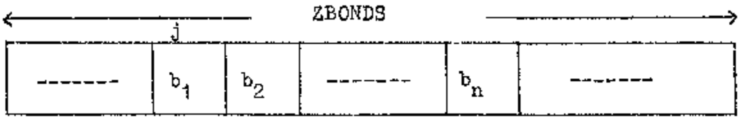
OR (b) Ø if the m^{th} block is the first block allocated for the array

J is either

(a) a forward pointer to the next block of YSTORE allocated for the same array

OR (b) Ø if the m^{th} block is the last block allocated for the array.

Diagram 1.2(b) : A ZSTORE entry for a block not on the free space list.



j is given by the sixth column of the dope vector entry

Diagram 1.2(c) : Shows a typical ZBONDS entry for an n-dimensional non-scalar variable.

ZBONDS is maintained in the following way. When a non-scalar variable is encountered, for example the variable A in

A ← 3 4 5

an entry is set up in DOPES and ZBONDS.

If A is redimensioned such that its number of dimensions is <u>increased</u>, for example, in the statement

A ← 2 4 ρ A ,

then the elements of the old ZBONDS entry are set to -1. A new entry is created for A, starting at position ZBPTR.

If A is now redimensioned such that its number of dimensions is <u>decreased</u>, the relevant part of the ZBONDS entry is updated and the remaining part set to -1's .

It can be seen that if an APL routine contains a number of redimensioning operations, (occurrences of the dyadic "rho" operator), the wastage of space in ZBONDS can become considerable.

A garbage collection mechanism enabling unused space to be retrieved was therefore devised. If there is insufficient space left in ZBONDS for a new entry to be created, ZBONDS can be scanned for entries with value -1. Such entries can be removed by shifting subsequent valid entries along the appropriate number of places to produce more free space at the end of ZBONDS. The appropriate dope vector entries must also be updated.

1.2.4 Accessing array elements

APL non-scalar variables are mapped onto the one-dimensional array YSTORE. Since the size of an APL array can vary dynamically, the array elements will not necessarily be stored in consecutive blocks of YSTORE.

The ZSTORE elements associated with each block of YSTORE contain both forward and backward pointers, as described in §1.2.2 . To access a previously stored vector or array element, the following strategy is required,

(a) a key is derived from the non-scalar variable name,

(b) the address of the dope vector entry for the non-scalar is determined (using (a)),

(c) the first ZSTORE element associated with the non-scalar is obtained (using (b)),

(d) the ZSTORE elements for the array are accessed in turn until the appropriate block is found,

(e) the index (in YSTORE) of the element to be accessed is found.

For large arrays it can be seen that a large number of ZSTORE elements may have to be accessed before the appropriate block of YSTORE can be located.

An enhancement of the above method would be to store the exact location (in YSTORE) of the last element accessed for a given array. Since consecutive access is most likely, it would thus be sufficient simply to move forward or backward from the position of the last element accessed. This additional information could be incorporated into the dope vector table.

Using the accessing method outlined above, the access time can be costly for large arrays. However, if the maximum amount of space required for storage of a non-scalar is known in advance, the non-scalar elements can be stored in consecutive blocks of YSTORE. A much simpler accessing method could hence be used for the non-scalar. The array mapping can be used to determine the relative position of an element in a non-scalar. The desired location can thus be found directly after applying steps (a), (b) and (c) above.

The faster method is dependent on more information being supplied initially by the user. This facility is provided as a user option and is discussed in greater detail in §1.4 .

Vector or array subscripts can themselves be expressions. Thus it is not usually possible to locate the exact position in YSTORE of a vector or array element during conversion. Instead, vector or array element references are replaced in the output code by function calls. These functions provide as a result either the value of the element being accessed or its index in YSTORE. It is necessary to know the YSTORE index (not the value) if an array reference occurs as the left argument of a specification operator. A number of functions were written to produce the above effect:

(a) FIND - produces the index in YSTORE for a numeric non-scalar variable access

(b) UVFIND - produces the value of a constant vector element as result

(c) IRFIND - produces the value of an intermediate result element

(d) EVFIND - used for accessing of empty vectors or arrays

(e) LFIND - used for accessing of literal non-scalars

(f) SCFIND - used for accessing scalars.

Operands (both scalar and non-scalar) can be accessed in a number of ways (see §1.2.5), and functions (b) to (f) above were written to provide generality with the function FIND. This function is described in §1.2.5 .

1.2.5 The function FIND

The function FIND is applied to the subscripts of the vector or array referenced. In the case of an entire array access, loops are set up to access each of the elements in turn.

Before production of a FIND call, therefore, code is produced to store the subscript values or expression code in the array ZINDX. The appropriate locations of ZINDX are accessed in FIND and a function applied to these elements to produce the required index in YSTORE.

APL allows nesting of subscripted expressions, and care must be taken to ensure that only the required values of ZINDX will be accessed during one call of FIND. This is done by maintaining a stack of pointers ZPOINT, having stack pointer ZPT. During any FIND call the array ZINDX is accessed only from the positions defined by ZPOINT (ZPT-1)+1 to ZPOINT (ZPT).

The following example serves to illustrate the type of code produced corresponding to a subscripted variable.

EXAMPLE 1(a).

Suppose the APL routine contains a reference to

$$A\left[I + 1\right]$$

where I is scalar. Then the generated code is of the form shown below. Since no attempt was made at optimisation during the code generation stage, the code is not very efficient. However, under certain circumstances, optimisation will be possible. This is discussed in greater detail in Chapter VIII.

```
 Z∅  =  ∅
ZB1  =  ZPOINT (ZPT)
ZPT  =  ZPT + 1

 ZINDX (ZB1 + 1)  =  I + 1
ZPOINT (ZPT)  =  ZB1 + 1   ---  since  A  is one-dimensional
CALL STARTS (A_NAMES, Z1, Z2, ZNC)
CALL FIND1  (--- A_NAMES --- Y_n ---)
```

The use of Z∅ is redundant in the above example, but is included to allow for the possibility of non-scalar subscripts, in which case it would be required for looping operations, (see Chapter VIII).

The subroutine STARTS uses the value A_{NAMES} (the index of A in NAMES, an array whose use is discussed in Chapter II) to provide information to be used in the call of FIND1. This information is discussed in

Chapter V, §5.2 . The subroutine FIND1 contains a call of the function FIND discussed previously. The variable Y_n contains the value of the required element A[I + 1] . The parameters of the function FIND are discussed later.

In generating subroutine calls two possibilities existed:

1. the subroutine calls could have no global variables (for example, ZPOINT, ZINDX) in the parameter list. COMMON statements would thus have to be inserted in the subroutine bodies. The same process can be applied to functions,

2. global variables could be included in the parameter list and all non-scalar globals given unit dimensions.

Method 1 is obviously more efficient from the point of view of parameter linkage. There are two advantages, however, of Method 2,

(a) if the dimensions of any global non-scalar require to be altered it is not necessary to change these in each subroutine or function containing a reference to the particular non-scalar;

(b) COMMON statements need not be used, and this facilitates conversion of APL to, for example, ALGOL or PL/1 rather than FORTRAN.

Throughout the entire text Method 1 will be assumed, as this gives greater readability of the generated code.

A second example showing the usefulness of the ZPOINT stack for nested subscripts is given below.

EXAMPLE 1(b)

The following code is generated corresponding to A[B ; C[D ; E]] where A and C are non-scalar; B, D, E are scalar.

```
                 Z∅  =  ∅
                 ZB1  =  ZPOINT (ZPT)
                 ZPT  =  ZPT + 1
             ZINDX (ZB1 + 1)  =  B
                    ┌
                    |  ZB2  =  ZPOINT (ZPT)
                    |  ZPT  =  ZPT + 1
     code           |
corresponding       |  ZINDX (ZB2 + 1)  =  D
      to            |
  C[D ; E]          |  ZINDX (ZB2 + 2)  =  E
                    |  ZPOINT (ZPT)  =  ZB2 + 2
                    |  CALL STARTS ($C_{NAMES}$ ---- )
                    |  CALL FIND1 (--- $C_{NAMES}$ --- $Y_n$ --- )
                    |  ZPT  =  ZPT - 1
                    └
                       ZINDX (ZB1 + 2)  =  $Y_n$
                       ZPOINT (ZPT)  =  ZB1 + 2
                       CALL STARTS ($A_{NAMES}$ --- )
                       CALL FIND1 (--- $A_{NAMES}$ --- $Y_{n+1}$ --- )
                       ZPT  =  ZPT - 1
```

Arrays A and C are distinguished in the FIND1 calls.

It can be seen that no information is lost after the C array reference has been handled. Code production for the A array reference is resumed in the normal manner.

The stack pointer ZPT is increased when the symbol [is handled and is decreased when the symbol] is handled.

The function FIND plays an important part in the handling of certain APL mixed functions. These are:

(i) the reverse function

(ii) the monadic transpose function

(iii) the reverse function (applied along the first co-ordinate)

(iv) the ravel function

(v) the rotate function

(vi) the dyadic transpose function

(vii) the rotate function (applied along the first co-ordinate)

(viii) the compress function

(ix) the expand function

(x) the take function

(xi) the drop function

(xii) the compress function (applied along the first co-ordinate)

(xiii) the expand function (applied along the first co-ordinate)

(xiv) the concatenate function.

The reason for grouping these operators together can perhaps best be explained by example.

Consider the following expression:

4 ↑ B + C

(where B and C are vectors).

An APL interpreter (i.e. one without an embedded "look-ahead" facility) would access all the elements of B and C during the '+' operation. All but 4 of these elements would later be discarded when '↑' was dealt with. There is thus an inherent inefficiency in the above expression. This inefficiency can be removed by applying a different type of accessing technique in the function FIND. To do this it is only necessary to apply a function to the required subset of ZINDX and then use the normal accessing method.

Consider also ⌽M, where M is a vector of n elements. To access the Ith element of ⌽M, the code

```
ZINDX (----) = I
```

is generated, followed by a call of FIND. However, in this case the contents of ZINDX (----) could first be changed to n - I + 1 and the normal accessing method used.

Similarly, all the functions in the above group can be handled by altering the values of the appropriate ZINDX elements and applying the normal accessing method. The function to be applied to ZINDX (i.e. the relevant part of it) to produce the desired type of accessing is determined by the first parameter of the FIND call.

In Chapter IV it is described how bracketing can be used to delimit the scope of an operator and thus remove inefficiences. Briefly, here

```
4 ↑ B + C
```

is bracketed as (4 ↑ (B+C)).

The scope of '↑' extends over the whole of (B + C). The appropriate type of accessing can be applied to B and C during the '+' operation to remove the necessity for accessing all the elements of B and C .

The scope of an operator could not be so easily defined if reverse polish techniques had been used in the translation, (see Appendix 5).

An element of the result of U \ V can be zero. This is indicated by setting the result of the FIND call to n , where n is one greater than the number of elements of YSTORE. Code is therefore produced to test the result of the FIND call for this condition.

If the ravel operator (monadic comma) is applied to a scalar, then a vector result is obtained. However, it is necessary to set up storage in YSTORE for the single element result. Such a result is indicated by having a negative value returned from the FIND call. The value returned is the negative of the index of the scalar in NAMES.

These tests are carried out immediately after the FIND call. They are present in each call of FIND1.

The functions UVFIND, IRFIND, EVFIND, LFIND and SCFIND, mentioned previously, were written to allow for all possible operand types in handling the 14 listed mixed functions.

The function FIND has the following parameters:

(i) the first parameter is

(a) ∅ if the normal accessing method is to be applied,

(b) 1 - 14, depending on which mixed function (of the above group) is to be handled;

(ii) the second parameter is

(a) ∅ for normal accessing <u>or</u> for a monadic mixed function of the above group <u>or</u> if the left operand is scalar (in which case the third parameter represents a value (not an index in the array NAMES)).

N.B. NAMES is the character array where the characters comprising identifier names are stored. It is described in Chapter II.

(b) the type value (see Chapter II) for the left operand if non-scalar (in which case the third parameter is the NAMES index for the left operand);

(iii) the third parameter is

(a) ∅ for normal accessing <u>or</u> for a monadic mixed function of the above group

(b) a scalar variable name or constant

(c) the NAMES index for the left operand (for dyadic mixed functions of the above group);

(iv) the fourth parameter is

(a) a scalar variable name or constant

(b) the NAMES index for the right operand (for dyadic mixed functions of the above group).

The FIND calls produced corresponding to B and C in 4 ↑ B + C are:

FIND (1∅, ∅, 4, B_{index})

and FIND (1∅, ∅, 4, C_{index})

respectively, where

B_{index} = the index for B in NAMES = B_{NAMES}

C_{index} = the index for C in NAMES = C_{NAMES}

1.3 <u>Essential Initial Information</u>

Information which a user must supply with his APL routine(s) falls into two categories:

1. The user must specify the output medium for storage of the target language. This is done by specifying a value for the variable IOPTON. The value must be provided in G12 format. Table 1.3(a) shows the values of IOPTON associated with particular output media.

IOPTON value	Output medium
∅	line printer
1	card punch
2	magnetic tape
3	magnetic disc
>3	line printer

TABLE 1.3(a) : Shows IOPTON values and the associated output media.

2. The user must supply a list of all the variables in his routine(s) which are non-scalar at their first occurrence. An indication must also be given of whether the variables are literal or numeric. The reason for this requirement is as follows. Suppose a non-scalar variable name is used as an APL function parameter. The type of the variable may not be made apparent inside the function body. The parameter may therefore be treated as a scalar (and incorrect code generated) unless the user explicitly declares it to be non-scalar.

The number of non-scalar variables being declared is first provided in I6 format. This is followed by a list of variable names in the format of Diagram 1.3(a). The zero indicates that no additional dimension or bound information has been supplied. The list is scanned and, corresponding to each non-scalar variable name encountered, code is generated to set up an entry in the dope vector table DOPES at run-time of the converted routine(s).

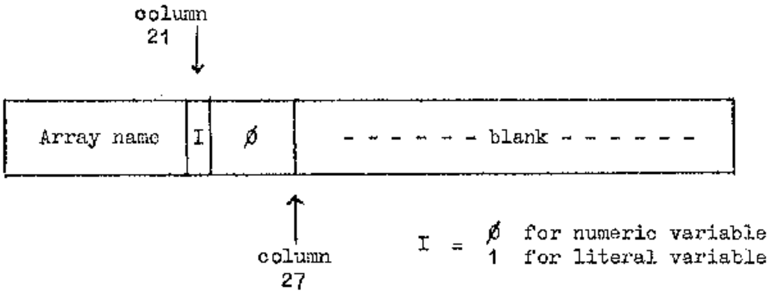


Diagram 1.3(a) : Shows essential information for non-scalars.

Initially, only one block of space is allocated for each non-scalar variable in the above list. This amount is increased or decreased as required during the running of the converted program. At this stage entries are set up in NAMES for all variable names appearing in the above list.

There are two cases in which an entry of the above form should not be supplied for a non-scalar variable. These are:

(i) If additional information is supplied, the entry will have instead one of the forms described in §1.4 .

(ii) If a variable is scalar initially and becomes non-scalar later, no entry of the above form should be supplied.

If no further information is provided for non-scalars, a certain amount of interpretation is essential. For example, to access an array element, a chain of ZSTORE elements must first be interpreted. If additional information is provided for non-scalars, the elements can be stored in contiguous blocks of YSTORE, thus reducing the amount of interpretation required.

Obviously, from the point of view of the execution time of the converted program, it is better to provide as much additional information as possible.

1.4 Additional Input Options

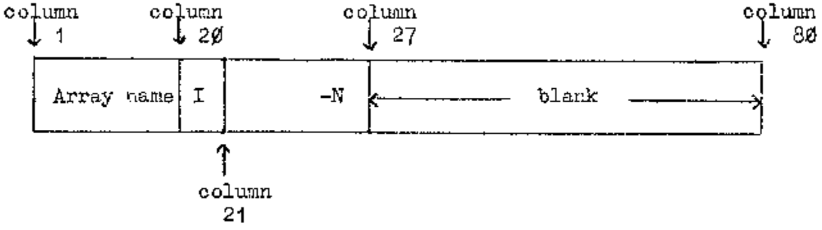
As discussed previously, it is to the user's advantage to supply as much information as possible regarding his non-scalar variables. The user may be able to supply full dimension and bound information for certain non-scalar variables at conversion time. For other non-scalar variables, however, he may only know the number of dimensions at this stage. It is possible that he will be able to supply the bounds for these variables at run-time of the converted routine(s).

Two additional input options are therefore available to the user. He can supply

1. the number of dimensions of a non-scalar variable with bounds for each dimension to be read in at run-time,
2. the number of dimensions of a non-scalar variable with *fixed* bounds for each dimension.

The information corresponding to forms (1) and (2) above should be provided in the format of Diagrams 1.4(a) and 1.4(b) respectively.

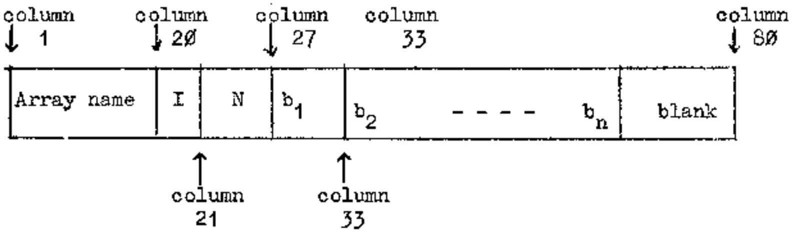
When bounds *and* dimensions are specified, these are assumed to be the maximum bounds for the array during running of the converted routine. Thus the maximum number of elements of the array is known. The number of dimensions and the bounds may differ initially from those supplied. Thus,



I = Ø for numeric variable
    1 for literal variable

N = the number of dimensions of the non-scalar variable

Diagram 1.4(a) : Form of additional information supplied for a non-scalar with the number of dimensions known, but not the bounds.



I = Ø for numeric variable
    1 for literal variable

N = the number of dimensions of the non-scalar variable

$b_1, b_2, ---, b_n$ are the bounds for each dimension

Diagram 1.4(b) : Form of additional information supplied for a non-scalar with both dimensions and bounds given.

from the initial information code is produced to set up only the first 4 parts of the dope vector entry.

If full information is provided, the non-scalar can be stored in contiguous blocks of YSTORE. This eliminates the need for the time-consuming access method used in the function FIND. The allocation of contiguous blocks of YSTORE could have been arranged at the time when initial information was processed. However, this would involve the insertion of an extra test in FIND. More time would thus have been required to access non-scalars for which no additional information was supplied. This is best avoided. A call of the function FIND is therefore produced for all non-scalar references and, where possible, this is replaced by a simpler accessing function as an optimisation process.

Storage of certain non-scalars in contiguous blocks of YSTORE is arranged in a pre-optimisation phase. It is done first for those arrays with full information given. With the bound information supplied at run-time of the converted routine(s), (i.e. after optimisation proper), the same process can be applied for non-scalars with only partial information supplied initially.

The following path is therefore taken.

(i) Read in initial information, process, store until (iii), set up NAMES entries and produce code to set up (partial) entries in DOPES.

(ii) Convert routine(s) to target language with FIND calls for every non-scalar reference.

(iii) Carry out pre-optimisation phase in which storage is arranged in contiguous blocks of YSTORE for those non-scalars with full information supplied.

(iv) Obtain bound information for the relevant non-scalars. Arrange these non-scalars in contiguous blocks of YSTORE.

(v) Replace FIND calls by simpler accessing function calls for all non-scalars with more than the minimum amount of information supplied.

(vi) Optimise the generated code.

(vii) Run the converted program.

Stages (iii) and (v) are discussed in Chapter VII. Stage (vi) is described in Chapter VIII.

At this stage an entry is set up in NAMES corresponding to each non-scalar variable name, and code is generated to produce partially filled dope vector entries. The rest of the information supplied is stored until required during the pre-optimisation phase.

The information temporarily stored at this stage is:

(i) the position of the non-scalar variable name in the initial list,

(ii) the index of the non-scalar in NAMES,

(iii) the dimension and bound information in its original form.

The first stage of the conversion proper is a lexical scanning phase, which is discussed in Chapter II.

The order of submission of information for the translation routines is given below.

1. IOPTON value (G12 format)
2. Number of non-scalar variables, N (I6 format)
3. N cards with information as described in §1.3 and §1.4 .
4. APL routine(s) to be converted
5. Blank card, signifying end of input.

CHAPTER II

LEXICAL SCANNING PHASE

An APL routine first undergoes a lexical scan. Each line of the routine is processed as described below, and the relevant information is stored temporarily on tape.

This scanning phase was initially introduced so that niladic function calls would be recognisable as such during subsequent processing. For example, consider the following routines:

```
∇ R ← A FN B ; X
      '
      '
      '
  X ← F + A
      '
      '
      '
      ∇

∇ R ← F
    '
    '
    '
    ∇
```

During processing of function FN, it is not known that F is a niladic function. This information only becomes available when the second function definition is encountered. Since the code generated depends on the types of all identifiers, it is necessary to scan each line in turn before the main line-by-line processing is carried out. This eliminates errors resulting from incorrect types being associated with identifiers.

The lexical scanning phase is generally useful as it simplifies the subsequent processes. In particular, it greatly simplifies the right-to-left scanning phase, which is discussed in detail in Chapter III.

The actions of the lexical scanning phase may be summarised as follows:

(i) All blank characters are removed.

(ii) When an identifier name is encountered for the first time, an entry is set up in the character array NAMES. The form of such entries for different identifier types is described in §2.2 . Thereafter all identifier names are replaced by the appropriate index in the array NAMES.

(iii) All other symbols not comprising identifier names are replaced by an integer value. Distinction is made at this stage between monadic and dyadic uses of particular symbols.

Each line of the APL code is scanned from left to right. Tests are first made for occurrences of the following symbols:

(i) the lamp-comment symbol

(ii) the 'del' symbol .

The actions carried out on recognition of these symbols are described in §2.11 and §2.12 respectively.

A test is then made for the occurrence of a symbol which can start an identifier name. When such a symbol is met, each character in turn of the identifier name is stored temporarily. After a complete identifier name has been decoded, the array NAMES is accessed. The method of accessing NAMES is also discussed in §2.2 . If no entry already exists in NAMES for the identifier name, a new entry is added to the end of NAMES.

The processed APL line is stored in the character array NOLINE. Corresponding to each identifier name, a 2-byte entry is added to NOLINE.

The entry represents the NAMES index for the identifier name. NAMES has 5ØØØ locations and, therefore, two bytes are sufficient to store the index for any identifier name.

A single entry is set up in NAMES corresponding to constant vectors, for example 3 4 5 in

X ← 3 4 5

Constant vector NAMES entries are discussed more fully in §2.1Ø .

The handling of other symbols is less straightforward. All symbols are distinguished initially with the aid of a symbol table, which is discussed in §2.1 .

The symbol table is arranged such that all dyadic operators are grouped together at one end, and all monadic operators are grouped later, with symbols which can be either monadic or dyadic appearing between. Letters, digits and special symbols follow the above three groups. Thus the address of a symbol in the symbol table can be used to determine the group to which the symbol belongs.

APL operators are later handled by the expansion of macros, as described in Chapter V. In general there is one macro for each operator, although a few operators (for example, +, –, x, ÷ , *) are grouped together and dealt with using a single macro expansion.

One method of handling each operator would be to replace the operator by a macro name and maintain a set of pointers giving the start address of each macro body. A more efficient method is employed here. Each operator has an associated macro number (<u>not</u> a name). The macro number is used to access a table, MCADDR, where the start addresses of the macro bodies are stored. Thus, for example, if '+' has macro number 21, then MCADDR (21) gives the start address of the macro body for '+' .

The above method eliminates the necessity to store a number of macro names in a table.

Operators are replaced in NOLINE by a 1-byte entry representing the required macro number. In fact, the entry gives the negative of the macro number, so that identifier and operator entries can be distinguished. (The second byte of an identifier entry may have a 1 in its left-most bit position {and thus be negative} but it will always be preceded by a positive entry. NAMES indices must be < 5000, which is $< 2^{15}$. Therefore the first part of an identifier entry will be positive.)

Identifier entries are stored in NOLINE with the two parts reversed, the reason being that the right-most (positive) part will be encountered first in the subsequent right-to-left scan.

Monadic and dyadic uses of the same operator are detected during this scan and the appropriate entries are generated in NOLINE. This is based largely on the fact that, if an operator is used in the dyadic sense, it will be preceded by an identifier or) or] .

A similar test is used to distinguish the use of the symbol '/' in u/v (where u is a logical vector) and f/x (where f is a dyadic operator). Two different entries are set up in NOLINE corresponding to '/' in the above expressions. Similarly for the symbol '⌿' .

Distinction is also made between the symbols '⎕' and '⍞' used for input or output purposes. If these symbols are used for output, they always precede a left specification arrow. A test is made for this occurrence. If the test is satisfied, then an entry is set up in NOLINE for '⎕' or '⍞' , but not for the left specification arrow. Thus,

□ ← A

generates entries in NOLINE for '□' and A only. The above expression is then regarded as the monadic operator □ operating on A .

If the test is not satisfied, then an input use of the symbols is intended. A different entry for □ or ⍞ would be set up in NOLINE for this case.

The symbol '.' is also used in a variety of circumstances. It can appear in

(i) a constant identifier name

(ii) an inner product

(iii) an outer product

The three uses are distinguished at this stage. In the case of outer products no entry is placed in NOLINE corresponding to the symbol '.' . The preceding symbol 'o' is sufficient to distinguish the occurrence of an outer product.

All the other symbols are replaced in NOLINE by an entry giving the negative of the appropriate macro number.

A table of information on APL symbols is given in Appendix 2. The method of distinguishing all the APL symbols is discussed in §2.1 .

Several values are stored on tape, together with NOLINE. These are values which are required in subsequent scanning phases. They include NOLPTR, which gives the number of entries in NOLINE for a particular APL line. Others are IFUNCT, IEXP and IFNI, whose functions are described in Chapter III, § 2.1 .

2.1 The Symbol Table and Its Method of Access

Symbols are first obtained in Z-code form. However, similar sets of symbols (such as the dyadic operators) cannot be grouped conveniently according to Z-code values. For this reason, a symbol table is maintained in which convenient sets of symbols are grouped together.

The symbol table is a one-dimensional array ISYMBT, 160 characters in length. It contains the Z-code representations of all the legal symbols in the APL language.

When a symbol is decoded a function is performed on the Z-code value. This produces the first address, I, to be accessed in ISYMBT. If the decoded symbol value equals ISYMBT (I), then the variable NADDR is set to I. Otherwise successive addresses of ISYMBT are accessed, starting from I , until there is a match. The correct address is then stored in NADDR.

Operators can be:

(i) dyadic
(ii) monadic
(iii) dyadic or monadic .

The group to which a particular operator belongs can be determined from the value of NADDR, for example:

(a) NADDR = 1 - 20 for purely dyadic operators
(b) NADDR = 21 - 38 for operators which can be either monadic or dyadic
(c) NADDR = 39 - 43 for purely monadic operators

In addition, the following groups can be distinguished:

(d) NADDR = 44 - 52 for delimiters

(e) NADDR = 53 - 120 for symbols which can start identifier names
(letters, Δ, $\underline{\Delta}$, digits, decimal point, overbar,
{high minus}, blank and quote)

(f) NADDR = 121 - 123 for remaining symbols (colon, del and locked del).

Within each of the groups (a) to (f), symbols appear in the symbol table in increasing order of Z-code value.

2.2 Identifier Names and the NAMES Table

A copy of all identifier names encountered is stored in the array NAMES. The identifier namo is thereafter replaced by the appropriate index in NAMES. The characters comprising identifier names can thus be re-accessed when required during the code production stage.

Identifier names must start with characters of the following types:

(i) a letter or a digit

(ii) a letter understruck

(iii) the characters 'Δ' or '$\underline{\Delta}$'

(iv) the characters '-' or '''

If any of these symbols is decoded, successive characters comprising the identifier name are stored in NAME, a 300-byte array. For literal identifiers, the enclosing quotes are first removed and double quotes inside the string are replaced by single quotes.

The elements of a constant vector are stored in NAMES with a blank character separating each element. A blank character also terminates each constant vector.

When the entire identifier has been decoded, the non-zero characters in NAME are compared in turn with the relevant parts of each NAMES entry of the same length and type. This process is repeated until either

(a) a blank entry is reached in NAMES, or

(b) a match is found between a NAMES entry and the contents of NAME.

The occurrence of (a) signifies that this is the first time the identifier name has appeared in the APL routine. A new entry is then set up in NAMES for the identifier. The form of the NAMES entry is given in Diagram 2.2(a). (The type of the blank entry reached should be tested as an empty literal vector will have a blank in the relevant part of the NAMES entry.)

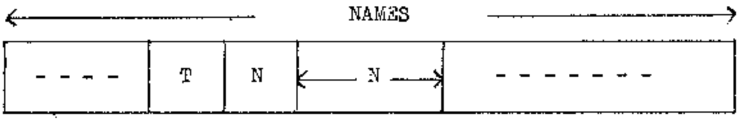
The occurrence of (b) indicates that the identifier name has already appeared in the routine. A previous occurrence of the identifier name is only confirmed if the type of the entry in NAME equals that of the entry in NAMES.

It can be assumed that all variable names start with the permitted characters, since only working APL routines will be converted.

Table 2.2(b) gives the possible type values for all the identifier types distinguished.

TYPE OF IDENTIFIER	TYPE VALUE
Non-scalar variable name (numeric)	1
Scalar variable name or constant	Ø
Literal (variable or constant, scalar or non-scalar)	-1
Function name	-2
Empty vector or array	-3
Label name	-4
Constant vector	-5

TABLE 2.2(b) : Shows type values for different types of identifier.



T = the type value associated with the identifier name

N = the number of characters in the identifier name

These two pieces of information are followed by the actual characters comprising the identifier name. T and N are in decimal ; the characters are in Z-code form.

Diagram 2.2(a) : A typical NAMES entry.

An APL identifier name can be from 1 to 77 characters long. Using the above method, only (k + 2) characters are required to store the identifier name, where k is the number of characters in the identifier name. This avoids the wastage of space which would result if the maximum number of characters was allotted for each identifier name.

Storage of the number of characters in an identifier name also makes it possible to scan quickly down NAMES to search for a particular identifier name. An identifier name is only stored in NAMES once, regardless of the number of times it occurs. However, two distinct entries would be set up in NAMES for the identifiers A and 'A' . The former would have a type value of ∅ or 1, and the latter a type value of ‾1 .

Constant vectors require no permanent storage in the array YSTORE. For such identifiers, the second element, N, of the NAMES entry gives the number of characters required to store both the constant and its associated blank entries. Constant vectors are again discussed in §2.1∅ . §2.3 to §2.10 describe the treatment of different types of identifiers.

2.3 Numeric Variable Names (Scalar and Non-scalar)

Diagrams 2.3(a) and 2.3(b) illustrate the NAMES entries which would be set up for the non-scalar variable name MARGARET and for the scalar variable name JACOBS respectively. Additional action is taken for the non-scalar variable name as described in Chapter I.

Certain identifier names are introduced during conversion of an APL routine. This is necessary, for example, in handling the looping operations implied by A+B, where either A or B (or both) is non-scalar. The arrangement is such that integer variable names introduced start with 'Z' ; real variable names start with 'Y'. Thus Z1, Z2, etc.

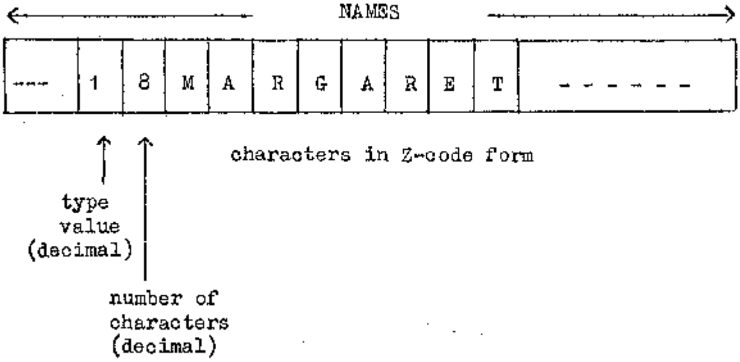


Diagram 2.3(a) : Shows entry for non-scalar variable name MARGARET in NAMES .

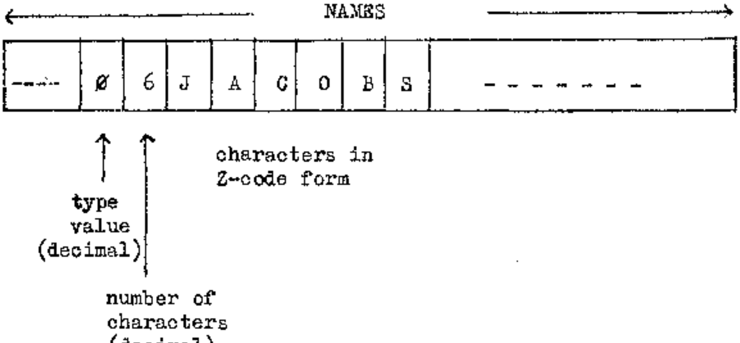


Diagram 2.3(b) : Shows entry for scalar variable name JACOBS in NAMES.

are used for integer variable names; Y1, Y2, etc., for real variable names.

In order to avoid duplication of existing variable names, the following strategy is employed. Variable names starting with 'Y' or 'Z' are altered to start with 'YY' or 'YZ' respectively.

FORTRAN variable names may not start with 'Δ' or 'Δ̲' or with a letter understruok. APL variables starting with these characters are therefore altered to start with 'YØ', 'Y1', and 'Y2<letter>' respectively. For example, A̲B would be altered to Y2AB. The name Δ̲ should be altered to Y1DEL, say, to avoid confusion with the generated real variable name Y1.

It is also necessary to shorten long variable names to comply with the rules laid down by the target language. At the same time uniqueness of identifier names must be retained. This is arranged in the following way.

A vector is set aside with one element to represent each letter of the alphabet. Each time a numeric variable name is decoded, an entry is set up in the vector. The element corresponding to the initial letter of the identifier name is set to 1. Thus, after scanning the entire APL routine (or set of routines) the vector is searched for zero entries. These entries give letters which have not been used to start identifier names. Such letters can then be used to start any shortened names. There will then be no confusion with existing names.

In the rare event of there being no zero entries left in the vector, it is still possible to scan NAMES for a combination of 2,3, etc., letters which have not been used to start identifier names. The search would stop when a unique combination of letters was found.

The method originally employed for reducing long names is outlined below. Consider, for example, the variable name A1234567A . If it is known that no identifiers start with 'B', then the name can be shortened to BA1234 without destroying the uniqueness criterion.

All further occurrences of A1234567A must be reduced similarly, and thus 'B' has to be associated with the identifier name in some way. However, the vector element corresponding to 'B' must now be set to 1, so that no other long names will be shortened to start with 'B', as this could also upset the uniqueness criterion.

This method requires one spare letter for every long name to be reduced. Thus, if only a few letters are available, these can quickly become exhausted. A method of avoiding this problem was therefore devised.

No reduction of long names can be done until after the lexical scanning phase, since spare letters will not be known until then. If any one letter has not been used to start an identifier name, then this letter can be used to start all the shortened names. For example, if 'X' is spare, then successive long names can be shortened to X1, X2, X3, etc.

A table is maintained associating each long name with the appropriate integer. This is done as follows:- If 'X' is spare, and X <n> (where <n> is any integer) is to replace A1234567A, then location <n> of the table will contain the index of A1234567A in NAMES. Entries can be set up in the table as the names are encountered.

This method is still unsatisfactory if no spare letters are available. However, a far greater number of cases can be handled before it is necessary to look for a unique combination of unused letters. The method can be made foolproof by reserving a specific letter, say X, to start shortened names and replacing each name starting with X by Y3X .... . This refinement has not been done at present, but it could be incorporated without much effort.

It should also be borne in mind that no non-scalar variable names are reproduced on the output stream. (Non-scalars are mapped onto YSTORE.) These can also be used (if not too long) to replace long names. The indices of the two names in NAMES would have to be associated. Long non-scalar variable names need not themselves be reduced.

2.4 Numeric Constants

The entry set up in NAMES for the constant 3.142 is illustrated in Diagram 2.4(a).

No restrictions are placed on numeric constants other than the practical limits set by the computer on which the converted routine is to be run. For example, on an IBM 36Ø machine an integer constant must have a value less than 2^{31}, since the word length of the computer is 32 bits.

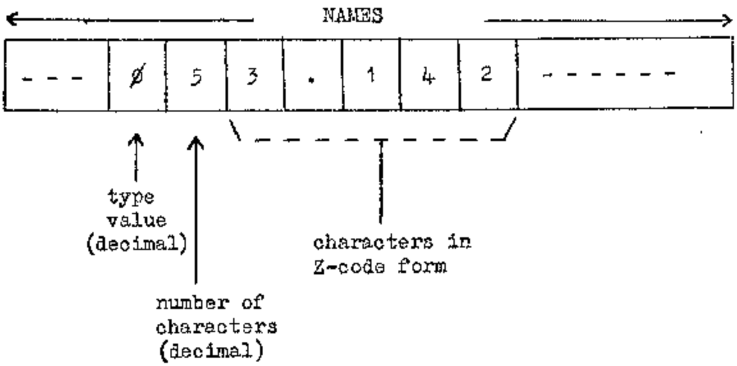


Diagram 2.4(a) : Shows NAMES entry for constant 3.142 .

2.5 Literal Constants

These are stored in NAMES with a type value of -1. For example, the NAMES entry for the literal constant 'AB' 'C' would be as shown in Diagram 2.5(a).

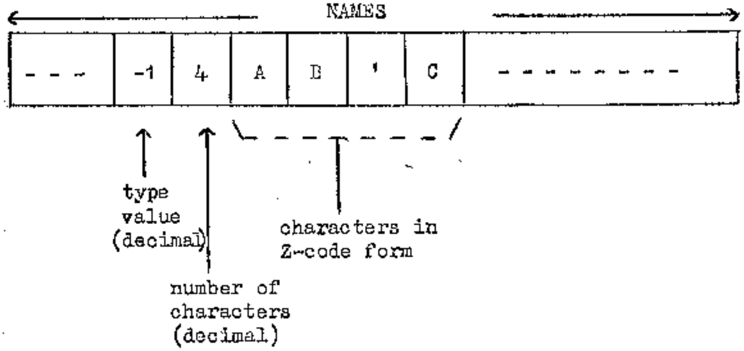


Diagram 2.5(a) : Shows NAMES entry for the literal constant 'AB''C' .

The enclosing quotes do not appear in NAMES and the double quotes have been replaced by a single quote.

The literal constant '' (signifying an empty vector) is treated similarly. The corresponding NAMES entry is illustrated in Diagram 2.5(b).

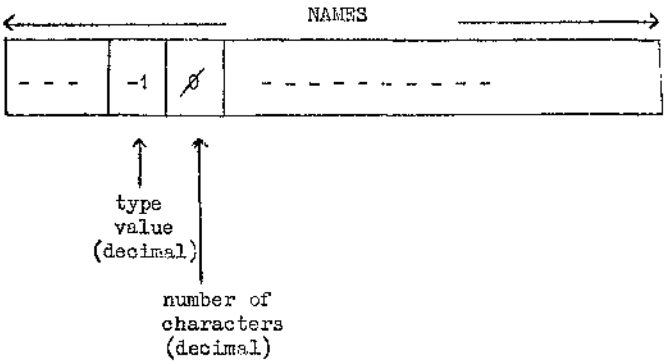


Diagram 2.5(b) : Shows NAMES entry for the literal constant '' .

2.6 Literal Variable Names

If the APL program contains a statement of the form

Z ← ⟨literal constant⟩

then Z is a literal variable name and its NAMES entry has a type value of ‾1 . However, it is necessary to distinguish between a literal variable name Z and a literal constant with value 'Z'. For this reason a two-dimensional table, LITBLE, is maintained. An entry in LITBLE provides the following information:

(i) the index (in NAMES) of the literal variable,

(ii) the index (in NAMES) of the literal constant currently associated with the literal variable.

LITBLE is accessed sequentially.

Suppose a function has to be applied to a literal variable. The index (in NAMES) of the associated literal constant can be obtained from LITBLE. Then the function can be applied instead to the appropriate constant to produce the required result.

The following process can be carried out to distinguish between literal variables and constants. First, test for an entry in column 1 of LITBLE equal to the index of the literal in NAMES. If no entry exists, the literal is a constant. Otherwise, it is a literal variable name. The second column of the LITBLE entry then gives the NAMES index of the currently associated literal constant.

No storage is set aside in YSTORE for literal vectors or arrays. These are stored in NAMES in row-major order. Entries are set up for literal non-scalars (at run-time) in the dope vector table DOPES and in the array ZBONDS, which contains bound information. The DOPES entry has two dummy values in columns 3 and 4 (since no storage is required for literals in YSTORE).

Suppose an APL routine contains a statement of the form

Z ← <literal constant 1>

and later there is a statement of the form

Z ← Z,<literal constant 2>.

Then, if <literal constant 2> is not equivalent to ' ', the new constant associated with Z requires a larger NAMES entry. It is thus necessary to

(a) create a new entry in NAMES giving the new value of Z,

(b) update column 2 of the LITBLE entry for Z to point to the new associated constant,

(c) update the ZBONDS entry for Z .

---	Ø	3	A	B	C	-1	4	-1	-1	-1	-1	1	2	X	Y

← NAMES →

Diagram 2.6(a) : Shows a possible structure of NAMES before garbage collection

----	Ø	3	A	B	C	1	2	X	Y	- - - - - - - - -

← NAMES →

Diagram 2.6(b) : Shows the corresponding structure of NAMES after garbage collection

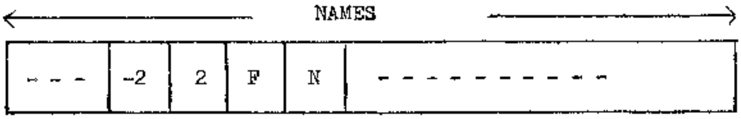
If the above process is repeated a number of times, a garbage collection mechanism may be needed to retrieve unused space in NAMES. Entries no longer required can be set to -1 . The second byte of header information (giving the number of characters in the identifier name) must, however, be retained. This is required so that NAMES will still be scanned properly.

The garbage collection procedure is only carried out if there is insufficient space left in NAMES to add a new entry. Entries containing -1's in the character parts can be removed and subsequent valid entries shifted along. It is also necessary to update the NAMES indices for valid entries which have been shifted along. Suppose, for example, that NAMES was set up as shown in Diagram 2.6(a) . Then, after garbage collection, the structure of Diagram 2.6(b) would be obtained.

A table is maintained to associate the correct NAMES index with the non-scalar XY and all subsequent valid NAMES entries.

2.7 Function Names

These are stored in NAMES with a type value of -2 . For example, the NAMES entry for the function name FN would be as shown in Diagram 2.7(a) .



- - -	-2	2	F	N	- - - - - - - - - - -

Diagram 2.7(a) : Shows NAMES entry for the function name FN .

The treatment of function definitions is discussed in §2.12 and in Chapters III and IV . Function calls are also discussed in Chapters III and IV.

2.8 Empty Vectors

These are produced as a result of expressions such as ⍳∅, ' ' , ⍴ <scalar >.

A variable name whose value is currently an empty vector is stored in NAMES with a type value of ‾3 . Suppose an APL routine contains code of the form

```
Z  ←  ⍳∅                                      ... (a)
   :
   :
Z  ←  Z,X                                     ... (b)
```

where X is an (m*n) matrix. Then, when (b) is handled, an entry has to be set up for Z in DOPES. The NAMES entry for Z must also be updated, that is, its type value should be changed from ‾3 to 1 . These changes take place at run-time of the converted routine.

2.9 Label Names

These are stored in NAMES with a type value of ‾4 . Each line in an APL routine has an implied label number associated with it. Entries are set up for these implied label numbers in NAMES. The method of associating a label number with every label name is discussed fully in Chapter VI . The label number is used in the output stream wherever the corresponding label name appears. This process is required because FORTRAN allows only label numbers, not label names.

2.1Ø Constant Vectors

These are stored in NAMES with a type value of -5 . Thus, for example, the entry in NAMES for the constant vector

3.1 4 2.39

would be as illustrated in Diagram 2.1Ø(a) .

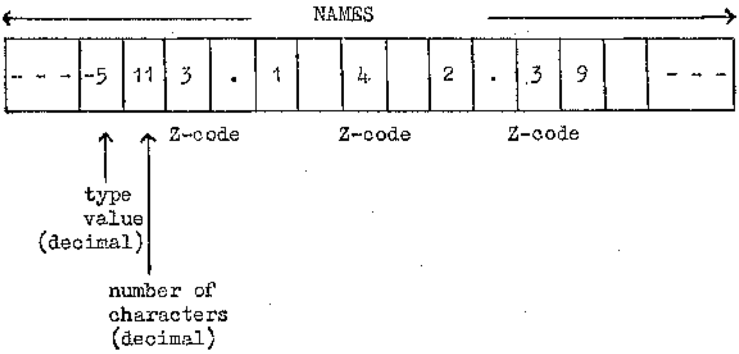


Diagram 2.1Ø(a) : Shows NAMES entry for the constant vector 3.1 4 2.39

The number of elements of (as distinct from characters comprising) a constant vector can be obtained by applying the function NOUNV to the index of the vector in NAMES.

2.11 Commentary

When a comment is encountered in an APL routine, a temporary entry is set up in NAMES. Such entries have a type value of -6 . After production of the comment line in the target language code, the NAMES entry for the comment is "removed" (by setting the relevant parts to -1's) .

A new NAMES entry is set up each time a comment is encountered, so that a comment entry in NAMES can be "removed" when completely handled without testing for occurrences of the same comment elsewhere.

A NAMES entry for a comment always has the number of characters part set to 79, i.e. the entire line (except the first character) is regarded as the operand for the monadic lamp-comment operator. Blanks in a comment line thus have no significance.

If necessary, garbage collection is used to retrieve space in NAMES. The method of retrieval is described in §2.6 .

2.12 Use of the 'DEL' Symbol

One of the first tests made on an APL input line is for the occurrence of a 'del' symbol at the left-hand end of the line. This can mean:

1. a function definition header statement,
2. a closing 'del' on a line by itself (signifying the end of a function definition).

For (2), a single entry is set up in NOLINE. This entry represents the negative of the closing 'del' macro number.

Function header statements deserve special mention. Entries are set up in NAMES for each identifier in a function header statement. The function or subroutine name has a type value of -2 .

If a local variable name occurs, the fact that the name is local to a specific routine is taken into account in setting up NAMES entries. For example, consider

```
∇ R ←— X  FN  B
        '
        '
        '
 A  =   X + F B
        '
        '
        '
        ∇

∇ R ←— F N ; X
        '
        '
        '
        ∇
```

The variable X in function FN is inaccessible during processing of function F . All references to X in function F are taken to mean the local variable name X .

This is simulated in the following way in setting up NAMES entries. During the lexical scan of function FN, one entry is set up in NAMES corresponding to X . During the lexical scan of function F, a new entry is set up in NAMES for the local variable name X and the previous occurrence is "locked". That is, it is inaccessible during the lexical scan of function F .

When the second NAMES entry for X is no longer required (after complete processing of the second closing 'del' symbol), the relevant parts are set to -1's in preparation for garbage collection. The second NAMES entry is thus "removed". The first entry for X must then be "unlocked".

Locking and unlocking is done as follows. When a local variable name is encountered, a new entry is set up in NAMES. All previous occurrences of the identifier name are locked by storing the appropriate NAMES indices in the array FNLOCS. During a scan of NAMES, FNLOCS is searched if a match is found. If an entry in FNLOCS equals the index for the matching entry, the search is resumed until an unlocked occurrence (one for which no entry exists in FNLOCS) is found. If no unlocked entry exists, then a new entry is added to NAMES.

To unlock an entry again, the corresponding entry in FNLOCS should be set to zero.

Now consider the example,

```
∇ R ⟵  A  FN  B ; X
       '
       '
       '
       ∇

∇ R ⟵  F  Y  ;  X
       '
       '
       '
       ∇
```

During the lexical scan of function FN , an entry is set up in NAMES for the local variable X . This entry is "removed" (by setting the relevant parts to -1's) when the first closing 'del' symbol has been completely processed.

Similarly, for the variable X in function F. After complete processing of the second closing 'del' symbol, no entries exist in NAMES for variable X . This is in accordance with the usage of the variable X.

Local variable name indices are stored as they are met in the array LOCAL. Thus, the appropriate entries can be "removed" later when they are no longer required. A second array, LOCS, is maintained to provide the number of entries to be "removed" at a particular time. For example, consider

```
∇ R ←— A FN B ; X ; Y ; Z
       '
       '
       '
       ∇


∇ R ←— F Y ; C ; D
       '
       '
       '
       ∇
```

Then LOCS(1) is set to 3 and LOCS(2) to 5. The difference between successive entries in LOCS gives the number of local variable names to be "removed" after complete processing of a particular closing 'del' symbol.

The locking and unlocking of local variables in this way is similar in concept to the use of the name-list table used in some ALGOL 6∅ implementations, (see Randell and Russell[4]).

A number of other values have to be stored to be re-accessed during the code generation stage. These are:

(i) For functions only (not subroutines) the result variable index and the array name index must be retained. These are stored in array FNIND. Consider the function

```
∇ R ←— A FN B
  R ←— A + B ∇
```

where R, A and B are scalar.

Then code of the following form is generated:

```
      FUNCTION  FN (A,B)                              ... (a)
       R  =  A + B                                    ... (b)
       FN  =  R                                       ... (c)
       WRITE (6,1ØØ) FN                               ... (d)
  1ØØ  FORMAT (1X, G 12.6)                            ... (e)
       RETURN                                         ... (f)
       END                                            ... (g)
```

By storing the appropriate variable name indices at lexical scan time, lines (c) and (d) can be produced at code generation time.

(ii) If a specification arrow is detected in a function header statement, an indication of this must be stored so that the correct code can be generated. For example, the code corresponding to

```
      ∇ A  FN  B
        R ←—  A + B ∇
```

would be of the form

```
      SUBROUTINE FN (A,B)
        R  =  A + B
        RETURN
        END
```

It is the absence of the left specification arrow from the function header which results in the generation of code different from lines (a) to (g) above. The array NEXP is used to retain an indication of the presence or absence of a left specification arrow until code generation (macro expansion) time. Entries on NEXP are:

(a) 1 for function header

(b) Ø for subroutine header.

Successive NEXP entries are accessed in turn during code generation, ensuring that correct code will be produced.

(iii) Now consider a subroutine with non-scalar parameters. For example,

```
∇ R  FN  A
  R ⟵  A + B ∇
```

where R and A are non-scalar, and B is a global scalar. Then the code generated is of the form

```
SUBROUTINE FN (ZF1, ZF2)
<Start of loops for array access>
    CALL FIND1 (---- ZF2 ---- $Y_n$ ----)
    $Y_{n+1}$  =  $Y_n$ + B
    CALL SPECS (ZF1, $Y_{n+1}$ ----)
<end of loops for array access>
    RETURN
    END
```

The subroutine SPECS handles the non-scalar specification. Corresponding to a call of the function FN, for example

```
C  FN  D   ,
```

code of the following form is generated:

```
ZF1   =   $C_{index}$
ZF2   =   $D_{index}$
FN (ZF1,ZF2)
```

where C_{index} = NAMES index for C

D_{index} = NAMES index for D

From the above it can be seen that the position of a parameter in the parameter list is important. The use of ZF1 or ZF2 is determined by position in the parameter list. The indices of parameter names are stored in the array FNPARM preceded by an integer giving the number of parameters (Ø, 1 or 2) . Thus, at the code production stage, after lexically scanning a set of routines, the position of specific parameters in a function or subroutine header can be obtained.

If a user of the converted subroutine FN, above, wishes to call FN with parameters C and D, a knowledge of the NAMES indices for C and D is required. The contents of the array NAMES are made available to the user. Care should be taken, however, to ensure that NAMES is not accidentally over-written. If the user can supply at conversion time a list of all the calls he intends to make of subroutine FN, then these can be treated as a main program and converted to FORTRAN automatically. This would remove the necessity for the user to access NAMES. Such action, however, will not generally be possible. Entries are not set up in NOLINE for left specification arrows, semi-colons or local variables appearing in a function header statement. Thus, for example, corresponding to

```
∇ R ← A FN B ; C ; D
```

entries would be set up in NOLINE for ∇ , R, A, FN and B only. Similarly for

```
∇ F X ,
```

NOLINE would contain entries for ∇ , F and X .

CHAPTER III

RIGHT-TO-LEFT SCAN AND PRODUCTION OF INTERMEDIATE CODE

At this stage the contents of NOLINE, together with certain other variables, have been stored on magnetic tape for each APL line supplied. The contents of NOLINE, corresponding to each APL line, are now processed in turn. The entire process involves:

(i) a right-to-left scan of NOLINE and production of intermediate code in the array NCODE,

(ii) a left-to-right scan of NCODE with frequent interruptions to expand macros,

(iii) generation of code using a series of macro expansions (the order of the expansions arranged in (ii)).

Stages (i) to (iii) are carried out for one line in entirety before the next line is considered.

A discussion of (ii) is deferred until Chapter IV, and of (iii) until Chapter V. This chapter describes stage (i) in detail.

The object of this phase is to separate the APL code into its component sub-expressions. This is done in such a way that all macro expansions (corresponding to specific operators) will be carried out in the correct order in the subsequent phase. The priorities of the APL operators will therefore be preserved, and code will be generated in the required order.

Since APL has a right-to-left system of operator priorities, this scan is carried out starting from the right. Brackets are introduced during this scan in such a way that the operator priorities are preserved. For example,

```
        A ←— B + C * D
```

is transformed to

```
        (A ←— (B + (C * D) ) )
```

during the right-to-left scan. In the subsequent left-to-right scan, therefore, the operators will be applied in the order *, +, ←— . The organisation of the macro expansions is described in Chapter IV.

At this stage the entire APL routine(s) has/have been lexically scanned, and the output stored temporarily. Each lexically scanned line is then re-accessed in turn, starting from the right. As each line is processed, an intermediate code form is set up in the array NCODE of 2ØØ characters.

The overall process so far is given in the flow-chart of Diagram 3(a).

Information may have to be added to either end of NCODE during the right-to-left scan, due to the insertion of bracketing. Hence, the production of intermediate code in NCODE starts near the middle and gradually extends outward. (The actual starting position is 16Ø, since most information is added to the left.) Two pointers LPTR and RPTR mark the limits in each direction and these are initially set to 16Ø.

The subroutine basic to the right-to-left scan is NCHAR. This stores the value of the next character in the variable LCHAR.

If LCHAR has a positive value, then it represents the left-most part of an identifier index. Suppose, for example, that the variable X has index 27 in NAMES. Then the two parts of the entry must be reversed before they are placed in the intermediate code. This is illustrated in Diagram 3(b).

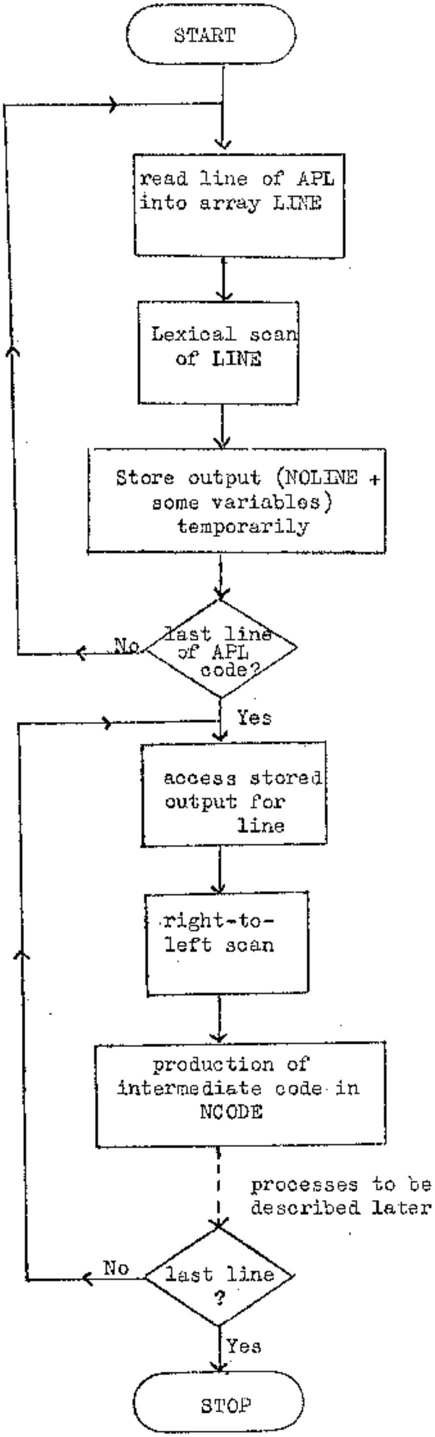

START
read line of APL into array LINE
Lexical scan of LINE
Store output (NOLINE + some variables) temporarily
last line of APL code?
No
Yes
access stored output for line
right-to-left scan
production of intermediate code in NCODE
processes to be described later
last line ?
No
Yes
STOP

Diagram 3(a) : Flow-chart of processes described so far.

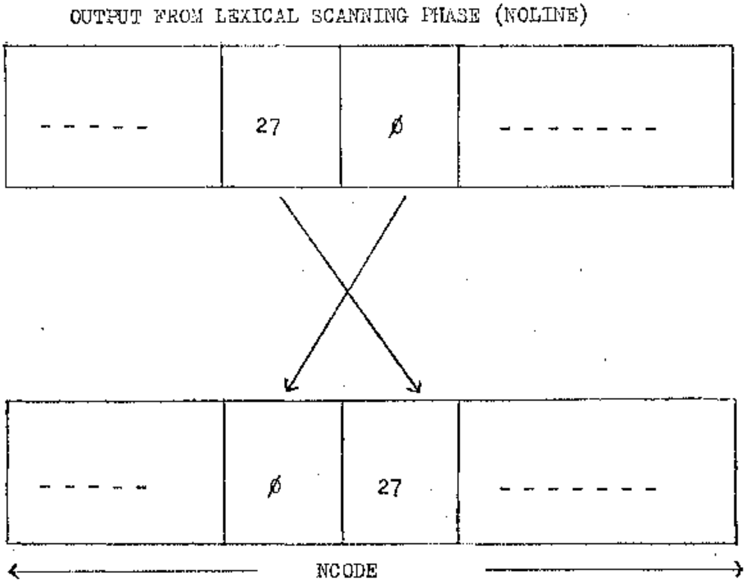


Diagram 3(b) : Shows the process carried out on an operand entry when transferred from NOLINE to NCODE.

§ 3.1 describes the treatment of a few special symbols during the right-to-left scan. Function and subroutine references are discussed in § 3.2 .

Operator entries are transferred straight from NOLINE to the intermediate code.

Three distinct types of bracketing are introduced:

1. bracketing of monadic operations,

2. bracketing of dyadic operations,

3. bracketing of function calls.

Thus, for example, the expression

A + B * - FN X

would be bracketed as

(A + (B * (- (FN X))))

However, it can be seen that the number of)'s on the right-hand side can be great. Hence, a single entry is used to replace a large number of brackets.

Opening round brackets are represented in NCODE by the negative of the macro number for (. Single)'s are represented similarly.

Macro numbers range from one to ninety-two. However, certain macro numbers (including numbers 84 onwards) do not correspond to specific operators, see Appendices 2 and 3. Eight bits are used to represent an operator entry in NOLINE and NCODE. Thus corresponding to macro numbers 1 to 83, there will be entries in NOLINE ranging from 173 to 255. The left-most eight bits of an operand entry will always have Ø in the left-most bit position, as NAMES has only 5ØØØ bits. No confusion will arise, therefore, if multiple closing brackets are represented in NOLINE and NCODE by (16Ø + n), where n is the number of closing brackets (up to a maximum of 12). More than 12 closing brackets can be represented by multiple entries of the form (16Ø + n), where $n \leq 12$.

3.1 A Few Special Symbols

Special care must be taken for some symbols. These are:

(i) [,] , (and)

(ii) ; inside square brackets

(iii) ; outside square brackets

(iv) □ and ⊡ , inner and outer products.

3.1.1 [,] , (and)

It is possible for a complete sub-expression, or set of sub-expressions, to appear within round or square brackets. Hence, it is necessary to leave a few locations of NCODE empty so that)'s can be inserted if required. Consider, for example,

(A + B * C) - E

The above expression is bracketed as

((A + (B * C)) - E) .

An extra closing bracket has been inserted here between C and) .

The number of brackets to be inserted is, of course, dependent on the APL statement, and space has been left to insert up to 6*12 closing brackets.

The values of the right and left pointers for NCODE are thus updated as follows:

RPTR = LPTR - 6 ... (a)

LPTR = RPTR ... (b)

(By altering (a) to RPTR = LPTR - N , space can be left to insert up to N closing brackets.)

The effect is represented pictorially in Diagram 3.1.1(a).

It is also necessary to store the original value of RPTR to enable subsequent closing brackets to be inserted in the correct place after the bracketed expression has been dealt with. For example, consider

$$A + (B * C + E) - D$$

The required bracketing is

$$(A + ((B * (C + E)) - D))$$

Production of bracketing is done in the following order:

(i) Obtain D and then - . Insert closing bracket after D for - .

(ii) Obtain) . Reset LPTR and RPTR as described above.

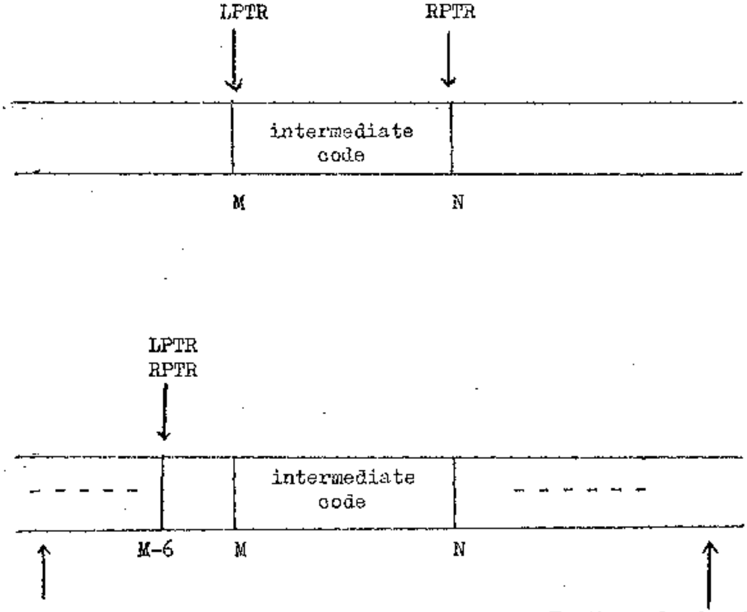
(iii) Obtain E and then + . Do not insert) after E as) is already present.

(iv) Obtain C and then * . Insert (for + before C and) between E and) . (Spaces have been left in NCODE.)

(v) Obtain B and then (. No need to insert (for * as (is already present. Reset RPTR.

(vi) Obtain + . Insert +(in NCODE. The (corresponds to bracketing of the - operator and its operands. Now insert) for left-most + symbol. Note that this) is inserted after D) and not after E), since RPTR has been reset to its previous value.

(vii) Obtain A . Now insert (A in NCODE for the left-most + .



Intermediate code will now be produced from position M-6 downwards, with brackets (possibly) inserted in locations M-6 to M-1

Further closing brackets may have to be inserted after position N when the whole bracketed expression has been handled.

Diagram 3.1.1(a) : Shows the re-positioning of the pointers LPTR and RPTR for NCODE when a) is encountered.

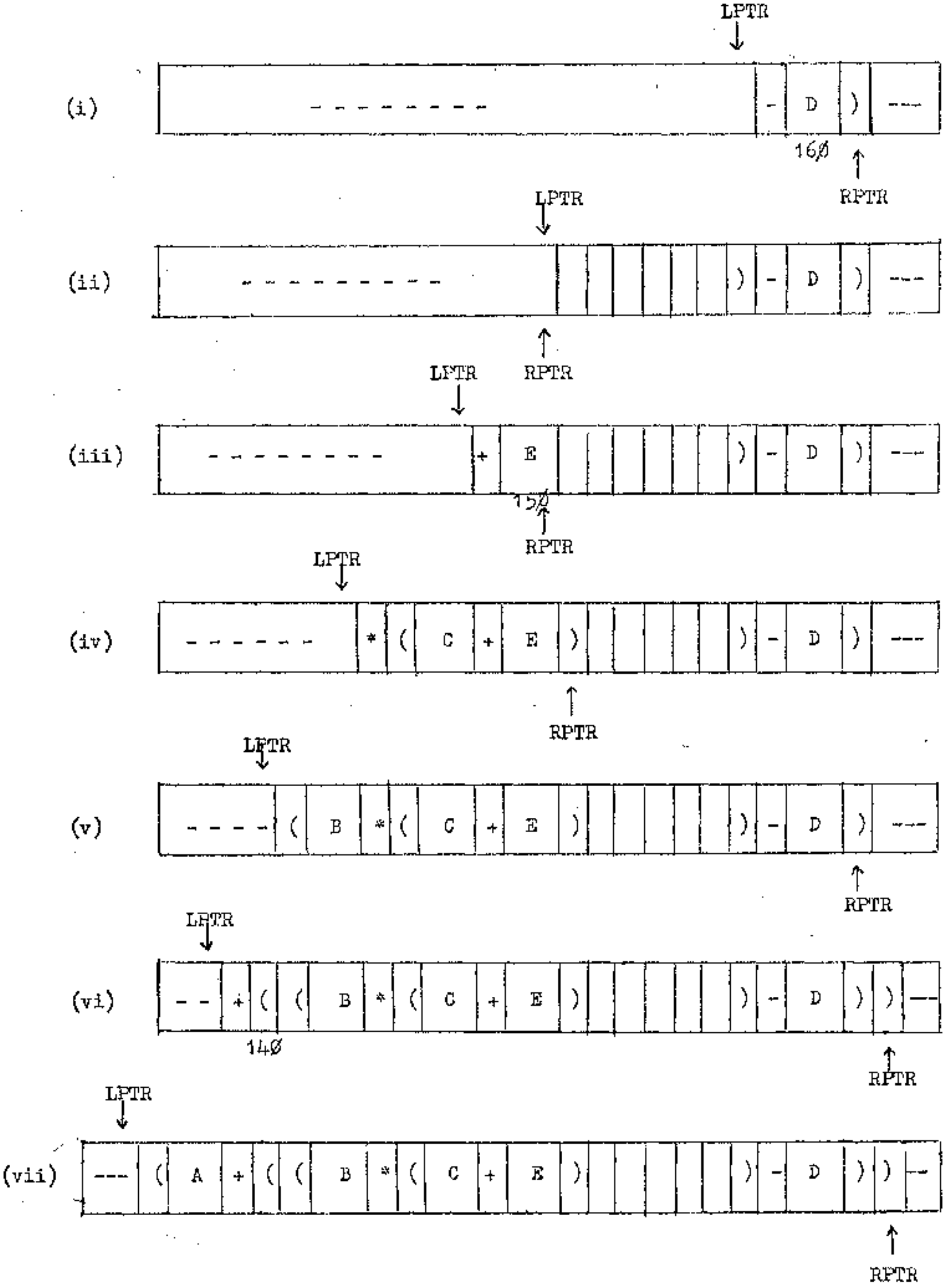


Diagram 3.1.1(b) : Shows stages of production of intermediate code for the expression

A + (B*C+E) - D

Diagram 3.1.1(b) shows the contents of NCODE for stages (i) to (vii) above.

In fact, a stack of RPTR values has to be maintained to allow for nesting of bracketed expressions. A value is stacked when) or] is recognised, and is unstacked when the corresponding (or [is obtained.

Any locations of NCODE left unused by the above method are set to 16Ø. This distinguishes them both from identifier indices (having value <127 in the left-most half) and the negatives of macro numbers. Such entries are ignored in the left-to-right scan of NCODE.

3.1.2 Semi-colons inside square brackets

Semi-colons are used inside square brackets to separate the subscript expressions. Since the subscript preceding a semi-colon can be an expression, it is again necessary to leave space for possible insertion of closing round brackets.

The number of semi-colons encountered in this way has to be counted so that the required number of RPTR values will be unstacked when [is obtained. The number of values to be unstacked in this case is (1 + K), where K is the number of semi-colons encountered.

Again a stack of semi-colon counts is maintained to allow for nesting of subscripted variables.

3.1.3 Semi-colons outside square brackets

Semi-colons are used outside square brackets to separate the components of a heterogeneous output statement. (Their other use, in function or subroutine definitions, is discussed in § 3.2) . Again it is necessary to leave space in NCODE to the left of the semi-colon to allow)'s to be inserted there. Consider, for example, C + A [B + C] ; 'IS THE RESULT'

The required bracketing is

(C + A [(B + C)]) ; 'IS THE RESULT'
↑
space required for insertion of) .

In this case, however, the right pointer is not reset when another semi-colon is obtained, as no further bracketing is required to the right of the semi-colon. Thus, the stack of RPTR values need not be updated. However, this stack is updated in the normal way when [and] are obtained.

3.1.4 □, ⍞, Inner and outer products

The handling of all these operators is much simplified by the action of the lexical scan.

□ and ⍞ have already been distinguished in their uses as input or output operators. Used for output, □ and ⍞ are now treated as ordinary monadic operators.

□ and ⍞ used for input are bracketed as operands, but stored as operators, (i.e. a 1-byte negative entry is placed in NCODE). For example,

B ⟵ □

is bracketed as (B ⟵ □) , whereas

⍞ ⟵ 'TITLE'

is bracketed as (⍞'TITLE') .

Inner and outer products have also been detected during the lexical scanning phase. Thus, for example, A + . * X and B ∘ . + Y are bracketed as (A + . * X) and (B ∘ + Y) respectively.

All other multiple uses of symbols have been distinguished during the lexical scanning phase and hence present no problems during the right-to-left scan.

3.2 Function and Subroutine References

These can be divided into two groups:

(i) function and subroutine definitions

(ii) function and subroutine calls.

The actions required for (i) and (ii) above are discussed in §3.2.1 and §3.2.2 respectively.

3.2.1 Function and subroutine definitions

If a function or subroutine definition is encountered during the lexical scanning phase, the variable IFUNCT is set to 1. The value of this variable is stored with the other lexical scan output to be reaccessed line by line. The values of the variables IEXP and IFNI are also available. IEXP has value 1 for a function definition header statement and value Ø for a subroutine definition header statement.

IFNI gives the number of variable names encountered in a definition statement, excluding local variables. Thus, for example, IFNI has value 3 for ∇ A FN B ; C and value 4 for ∇ R ← X FX Y .

The treatment of local variables in a function or subroutine definition statement has been described in Chapter II, § 2.12 . Local variable names are not present in MOLINE (corresponding to a header statement). Similarly, the symbols ← and ; have been removed.

Thus, it is only necessary to insert brackets round the remainder of the expression. The symbol 'del' used in function or subroutine definitions is thereafter treated as an operator with a variable number of operands. For example,

```
∇ A ⟵ B FN C ; D
```

is bracketed as

```
(∇A B FN C )
```

while,

```
∇ F X
```

is bracketed as

```
(∇F X ) .
```

The variable IEXP is tested during expansion of the opening 'del' macro to determine whether the code

```
FUNCTION - - -
   or
SUBROUTINE - - -
```

has to be produced.

3.2.2 Function and subroutine calls

These are treated as multiple operands and are bracketed during production of the intermediate code. For example,

```
C ⟵ A + FN X
```

is bracketed as

```
        (C  ←—  ( A + (FN X ) ) )
```

No distinction is made at this stage between function and subroutine calls.

3.3 Function Bodies and Function Parameters

Function or subroutine bodies are treated in the normal way except that the variable IDEL is set to 1 if a closing 'del' is obtained. It does not appear in the intermediate code, but is tested when a line has been completely processed to determine whether the closing 'del' macro has to be expanded.

If a function parameter is itself an expression, the expression is dealt with in the usual way. Thus, for example,

(i) (A+B) FN C is bracketed as ((A+B) FN C)

(ii) A FN B+C is bracketed as (A FN (B+C)) .

In the case of (i) it must be remembered that the number of identifiers obtained before (A+B) was 2. This is necessary to keep the bracketing correct. To organise this, a stack is maintained with values either ∅ or 1, depending on whether the bracketed expression is the left parameter of a function call. The stack is necessary to handle nesting of bracketed parameter expressions. For example,

```
        ( (A+B) FN C ) F  X .
```

A similar situation applies for subscripted left parameters.

3.4 An Example

To complete Chapter III, an example is given showing the conversion of a line of APL code to intermediate code form.

EXAMPLE 3.4(a) .

Consider the APL statement

A ⟵ B[C+D*E] FN X .

The APL line is first read into the array LINE and then scanned from left to right. During the lexical scan, entries are set up in NAMES and NOLINE as described in Diagram 3.4(a) .

It has been assumed that

(i) B is non-scalar, FN is a function name and all the other variables are scalar,

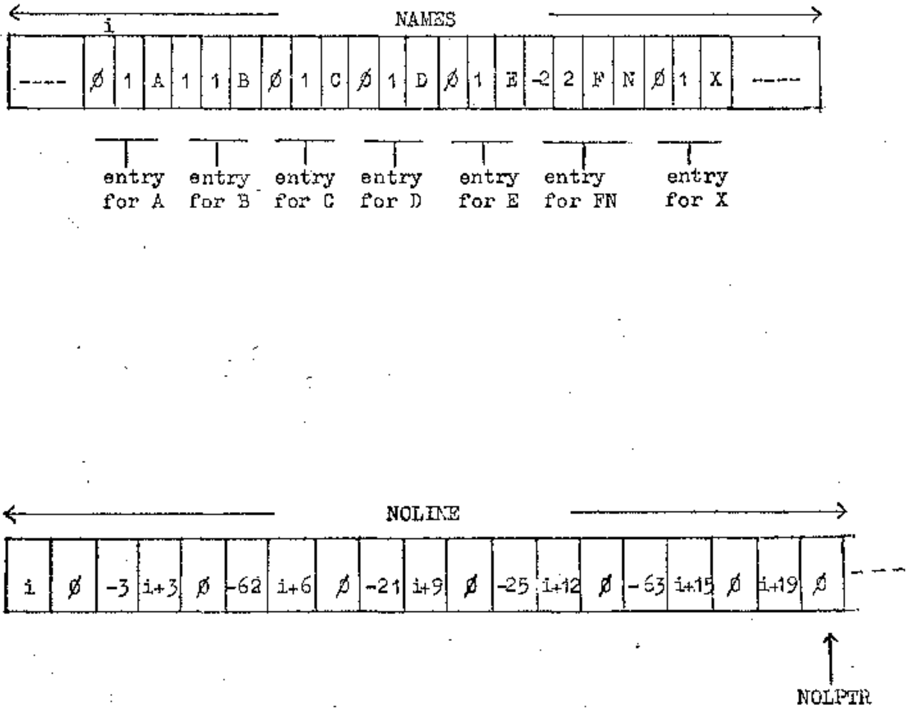
(ii) i + 19 < 128 (making the other part of the operand entry zero).

The variable NOLPTR now has value 19. The array NOLINE, together with the values of variables NOLPTR, IFUNCT, IEXP and IFNI are now stored until the entire source input has been lexically scanned.

When reaccessed later, the array NOLINE is scanned from the right and the intermediate code shown in Diagram 3.4(b) is generated in NCODE.

In fact the variable names would not be stored sequentially in NAMES as some of these entries must have been encountered previously.

A comparison of the bracketing method and the reverse polish method is given in Appendix 6.



The most significant character of the operand entries in NOLINE is at the right.

Diagram 3.4(a) : Shows the entries set up in NAMES and NOLINE for the expression

A ⟵ B [C+D*E] FN X

14Ø

---	-64	Ø	i	-3	-64	Ø	i+3	-62	-64

15Ø

Ø	i+6	-21	-64	Ø	i+9	-25	Ø	i+12	162

16Ø

					-63	Ø	i+15	Ø	i+19

17Ø

+162									

Diagram 3.4(b) : Shows the entries set up in NCODE corresponding to

A ⟵ B [C+D*E] FN X

CHAPTER IV

LEFT-TO-RIGHT SCAN, PRODUCTION OF STACKED INFORMATION AND ORGANISATION OF MACRO EXPANSIONS

The object of this phase is to expand a series of macros, the order being determined by the brackets inserted during the previous scan.

Macros are mainly expanded on recognition of a closing round bracket, although there are a few macros which require immediate action. These immediate action macros are discussed in detail in §4.3 .

During this phase also, some information is produced on a stack. The stack is accessible from the macro bodies by means of the macro instruction ?<n> , where <n> is any integer $\leq$ 5ØØ . ?<n> accesses the <n>th position of the stack, starting from the current base level. The organisation of the stack is described in §4.1 .

The character array NCODE contains the intermediate code produced during the right-to-left scan. This code is now scanned from left-to-right and information relating to operands and operators is stacked in the manner described in §4.1 . The pointers RPTR and LPTR define the positions to be accessed during this scan.

The symbols (and) are used to bracket 3 distinct types of expressions. These are:

(i) dyadic operator sub-expressions

(ii) monadic operator sub-expressions

(iii) function calls.

As was discussed in Chapter III, all operators have been replaced by by the negatives of their macro numbers and all operands by the indices of their NAMES entries.

In the case of (iii) above, the operand indices appear in the same order as the identifiers were used in the source text. The operands could, of course, be expressions, in which case they would fall individually into one of the categories (i) to (iii) above.

Sub-expressions of types (i) to (iii) can be combined in any order.

Groups (i) and (ii) can be distingushed by examining the macro number. Group (iii) is distinguished from the others by the absence of an operator entry between the brackets. The organisation of macro expansions is described in §4.2 .

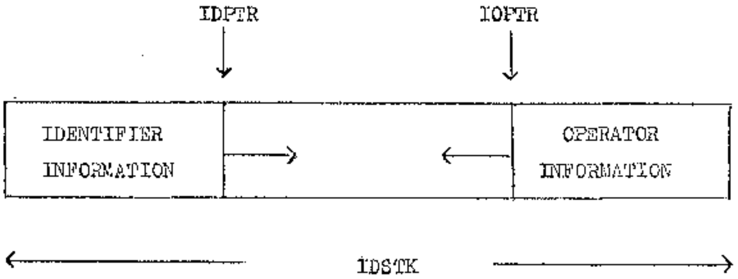
4.1 Organisation of Stacked Information to be Used at Macro Expansion Time

Information relating to operands and operators is stored on a double-ended stack IDSTK, having 500 half-words. Entries relating to identifiers are stored at one end of IDSTK and information pertaining to operators at the other. This is illustrated in Diagram 4.1(a) .

With the exception of operators requiring immediate action, an entry is placed on IDSTK each time an operator or delimiter entry is detected in the scan of NCODE. The entry to be stacked is the macro number for the operator or delimiter.

When an identifier entry is recognised during the left-to-right scan, the NAMES index for the identifier is stacked on IDSTK.

In general, there will be nesting of the sub-expressions corresponding



IDPTR is stack pointer for the operand end of IDSTK

IOPTR is stack pointer for the operator end of IDSTK

Diagram 4.1(a) : Shows the method of storage of information on IDSTK.

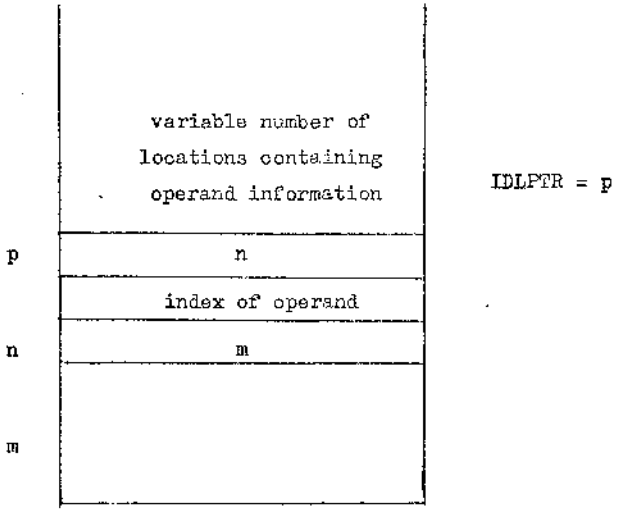


Diagram 4.1(b) : Shows a possible stack formation for operand entries in a nested sub-expression.

to an input APL line. Thus, there may be several sets of operand and operator information on IDSTK at any time. This causes no confusion for operators, as it is only necessary to stack a new operator entry when required. However, confusion can arise in the case of operand entries, for the following reasons.

Suppose operand and operator entries are stacked as they occur until a closing round bracket is detected. A series of macros is then expanded. There is an "operand" macro which, when expanded, will produce the required code for an operand, depending on its type value. The index of the operand from which code is to be generated by the operand macro expansion has previously been stacked on IDSTK, together with some further information relating to the operand. This information is discussed in § 4.1.1 and may involve a variable number of locations of IDSTK.

If a dyadic operator sub-expression has been decoded, then expansion of the operator macro is preceded by two expansions of an operand macro, one for each operand. For monadic operator sub-expressions there is one expansion of an operand macro followed by expansion of the appropriate operator macro. (Bracketed function calls are discussed in § 4.2 .)

When code has been generated for an entire sub-expression, operator entries are merely unstacked from IDSTK. Operand information is replaced by information for the result. Since the amount of information to be replaced is variable, confusion can result in the placing of the result information. This is to be avoided, as the result information will be used as operand information during any further expansions of the operand macro. The method of avoiding such errors is discussed below.

A pointer IDLPTR is maintained to enable only the relevant part of the operand information to be accessed at any time. IDLPTR is initially zero. The start of a new nested sub-expression is detected by the presence of an opening bracket in tho intermediate code. The method of separating the new sub-expression from the previous one is as follows:

1. the value of IDLPTR is stacked in the operand part of IDSTK

2. IDLPTR is then updated to the value of IDPTR.

Thus, if IDSTK (IDLPTR) is accessed later, the previous value of IDLPTR can be obtained. A set of backward pointers for IDSTK is thus produced. These pointers define the start of the relevant information for specific sub-expressions. The appropriate operand information is thus always obtainable at macro-expansion time. The starting position for the placing of result information is also known.

After a complete sub-expression has been dealt with, IDLPTR is set to its previous value (given by IDSTK (IDLPTR)). The stack pointer IDPTR is first reset to the old value of IDLPTR and the result information placed in this position. The stack pointers are reset and the result information stacked in the following order:

1. IDPTR = IDLPTR
2. IDLPTR = IDSTK (IDLPTR)
3. IDSTK (IDPTR) = < result information >.

This is illustrated in Diagram 4.1(b) . After dealing with the sub-expression whose operand information is on top of IDSTK, the pointers IDPTR and IDLPTR are reset. IDPTR is set to P , the value of IDLPTR, and the result information is placed in IDSTK (P) . The pointer IDLPTR is reset to n , given by IDSTK (IDLPTR). Thus operand information will continue to be added from the current base level of the stack (now n) .

A further illustration is given in §4.6 , where a complete example is worked through.

4.1.1 Information stacked before the operand macro is expanded

Operands are handled in the following way at macro expansion time. The NAMES index for the operand is obtained from IDSTK. This index is used to provide more information relating to the operand. The form of the information varies for different types of operands, as discussed below. All information is placed in consecutive locations above the current base level of IDSTK.

4.1.1.1 Scalar operands, label names, niladic function names, numeric non-scalars, literals and empty vectors

The information stacked for operands belonging to this group is described in Diagram 4.1.1.1(a) .

IDSTK (IDLPTR + 1) contains the index of the identifier in NAMES. This enables the identifier name to be reproduced on the output code.

IDSTK (IDLPTR + 2) contains the type value for the operand. This enables all the members of the group to be distinguished.

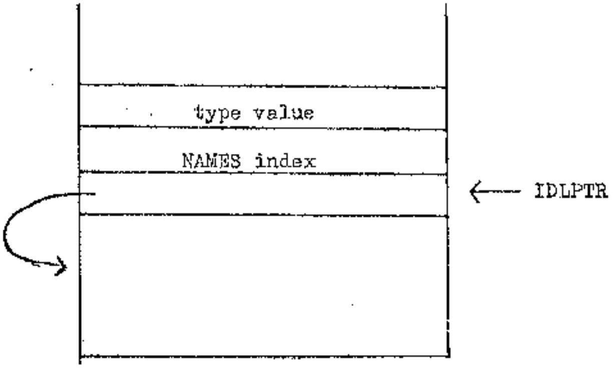


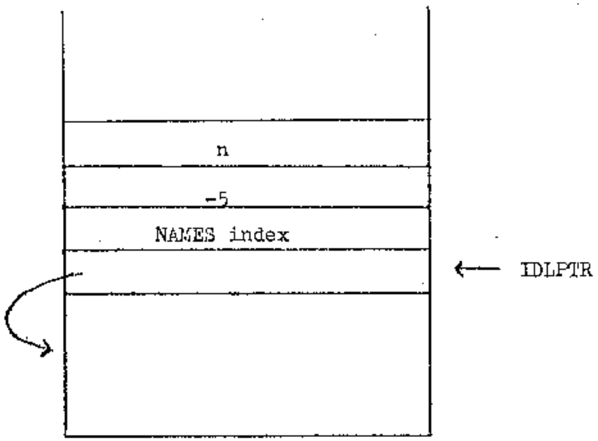
Diagram 4.1.1.1(a) : Shows the form of operand information stacked for scalars, labels, niladic functions, numeric non-scalars, literals and empty vectors.

4.1.1.2 Constant vectors

The information stacked for operands belonging to this group is illustrated in Diagram 4.1.1.2(a) .

IDSTK (IDLPTR + 1) again contains the index of the operand in NAMES and IDSTK (IDLPTR + 2) provides the type value.

IDSTK (IDLPTR + 3) contains the number of elements of the operand. This is the only case in which the bounds for a non-scalar are known at this stage.



n = the number of elements in the constant vector.

Diagram 4.1.1.2(a) : Shows the form of operand information stacked for constant vectors.

4.1.1.3 Intermediate results

The simple structure illustrated in Diagram 4.1.1.3(a) is used for storage of intermediate result information.

The entry -1∅ deserves special mention. For all other types of operands, a NAMES index is stacked on IDSTK. However, intermediate results are not stored in NAMES and they may appear as operands. A negative entry is used in place of a positive NAMES index to distinguish intermediate results from all other operand types.

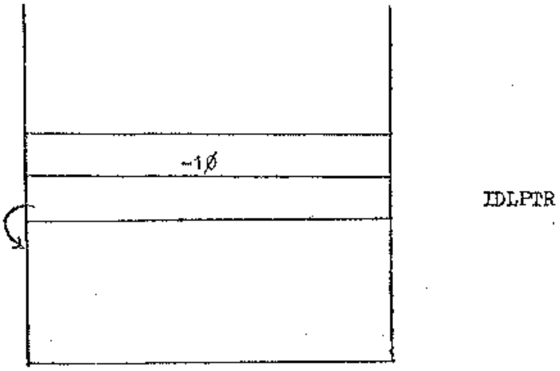


Diagram 4.1.1.3(a) : Shows the form of operand information stacked for intermediate results.

4.1.1.4 Monadic function and subroutine references

The information stacked for such operands is illustrated in Diagram 4.1.1.4(a) .

IDSTK (IDLPTR + 1) contains the NAMES index for the function name. Successive locations contain information for the function parameter, these entries taking one of the other forms described. (No new base level is created between the function name index and the parameter information.)

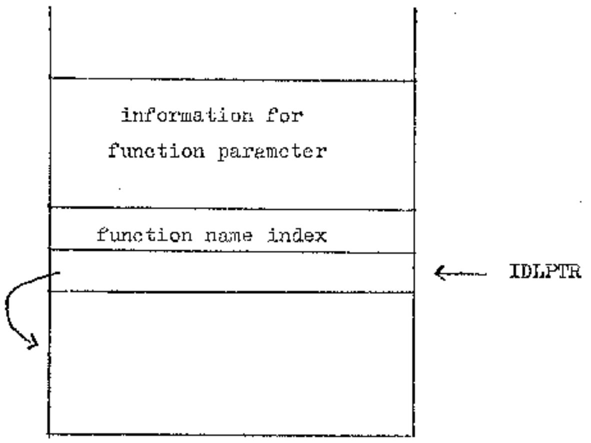


Diagram 4.1.1.4(a) : Shows the form of information stacked for monadic function and subroutine references.

4.1.1.5 Dyadic function and subroutine references

The information stacked for such operands is illustrated in Diagram 4.1.1.5(a) .

For operands belonging to this and the preceding group, the stacked information is used to produce a function or subroutine call.

4.1.1.6 Quad and Quote-Quad input

This case has been included for generality. Consider the APL statement

A ← B + □

Here the "operands" for + are B and □ . The □ symbol is an indication that the right operand for + is to be obtained at run-time. This is

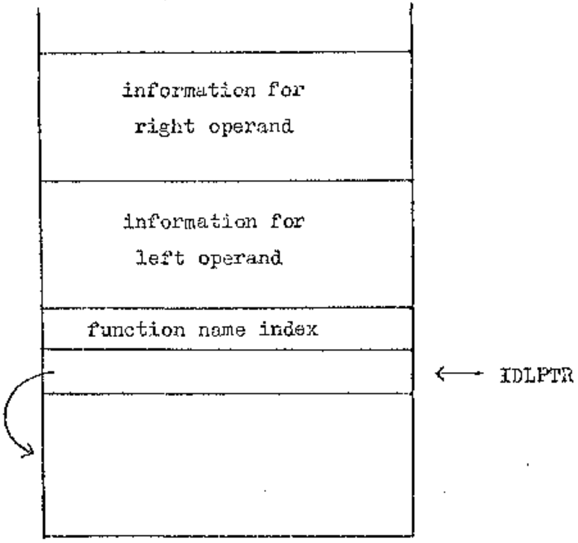


Diagram 4.1.1.5(a) : Shows the form of stacked information for dyadic function and subroutine references.

handled by placing a -2 entry on IDSTK corresponding to □ .

When the + operator is to be handled the operands are dealt with first and the -2 entry is detected. This indicates the presence of quad input and the appropriate macro is expanded to read in data. The data obtained is then used as the right operand for + .

Similarly, quote-quad input is indicated by placing a -1 entry on the identifier part of IDSTK.

4.2 Organisation of Macro Expansions

It is necessary to distinguish between bracketed sub-expressions and bracketed function or subroutine calls at macro expansion time. The following method is used.

For each level of nesting an indication has to be stored of the presence or absence of an operator between brackets. A character stack IBITS, with stack pointer IBIT, is therefore maintained.

Since an opening bracket indicates a new level of nesting, IBIT is incremented by 1 when (is recognised.

If an operator entry is encountered, IBITS (IBIT) is set to .TRUE. . When a closing round bracket is obtained IBITS (IBIT) is tested. If its value is found to be .TRUE. , a sub-expression involving an operator has to be handled. Otherwise a function or subroutine reference has to be produced. After dealing with the bracketed expression, return is made to the previous level of nesting and IBIT is decremented by 1.

When output code is being produced, it is accumulated on an array MTEMP of 8Ø characters. If a line of code has been completed, the contents of MTEMP are transferred to the output stream. Intermediate result code generated at macro-expansion time is stored in the array ITEMP. It can thus be re-obtained to be used as an operand for the next operator macro to be expanded. MTEMP can also be used to accumulate the next line of code.

ITEMP is a character array of 4ØØ bytes, and it is used to store the code corresponding to each operand. There is a pointer IBPTR, which is used to chain down ITEMP to obtain the required operand. Only the last two operands need to be accessed at any stage, since an APL operator has a maximum of 2 operands.

ITEMP (IBPTR) points to just before the start of the right-most operand. This address in turn points to just before the start of the previous operand. The data structure used is described pictorially in Diagram 4.2(a) .

IBPTR is thus a pointer for a backward chain, which enables the operands to be re-accessed when required. Using the above method of storage and access for operands, it is immaterial whether the operand is an identifier or an intermediate result.

When an intermediate result is placed in ITEMP, a -1∅ entry is stacked on IDSTK.

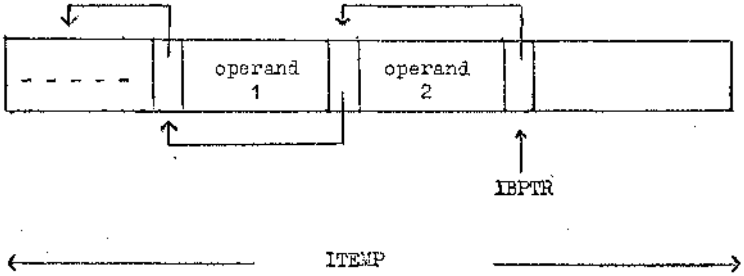


Diagram 4.2(a) : Shows the method of storage of intermediate results in ITEMP.

For a sub-expression involving an operator, the usual procedure to be carried out when) is obtained is described below.

1. The appropriate operand information is stacked as described in § 4.1 .

2. An "operand macro" is expanded. This uses the stacked information to produce the code corresponding to the operand and store it in ITEMP.

Use of an operand macro to produce code for operands removes the necessity to test identifier types in the operator macros. The operator macro bodies are consequently much simpler. Two operand macros are used, depending on the operators to be handled. These and other macros are discussed in Chapter V .

3. Steps 1 and 2 are repeated for a dyadic operator sub-expression.

4. The appropriate operator macro is now expanded. The required operands can be reproduced on MTEMP by use of the macro instructions LO and RO . These provide the left operand and right operand respectively. The compound macro instructions PL and PR also access the operands and produce code of the form

   Y< integer > = operand .

   These are also discussed in Chapter V .

5. The code for the result is stored on ITEMP and the pointers IBIT, IDLPTR, IDPTR and IOPTR are updated.

6. A -1Ø entry is stacked on IDSTK and the left-to-right scan continues.

For a function or subroutine reference, only steps 1,2,5 and 6 are carried out.

At each stage, the current bounds for the result are stored in the array ZCBNDS, from positions 1 to ZCPTR. Since the bounds can be updated dynamically, code is always produced to update ZCBNDS at run-time. This gives the user a check on the current bounds of his program at each stage during its execution.

The type of the result may also vary dynamically (for example, in the case of the statement X ← ⎕) .

The variable MARKER is used to denote the type of the result. The possible MARKER values are the type values for the seven types of operand distinguished.

The uses of the symbols [and] in expressions such as

(a) A [1] and

(b) + / [1] X

are distinguished in the following way. In (a), the symbol [is immediately preceded by an operand (it could also be preceded by a closing round bracket, for example, in (L,'ABC') [1]) . This is not the case for (b).

A variable ILEFT is used to distinguish (a) and (b) . This variable is continually being updated during the left-to-right scan. It is set to 1 when an operand index is encountered and reset to Ø when an operator entry is recognised. Thus the value of ILEFT can be tested when '[' is obtained to determine its use.

The action required for cases (a) and (b) is described in § 4.3 .

For most operators, the left operand is handled before the right operand, which will therefore be the right-most entry in ITEMP when the operator macro is expanded. There is, however, one notable exception to this rule. In the case of the left specification operator, the right operand must be handled first. This is to ensure that the correct type and dimension information will be associated with the left operand. For example, in the statement

X ⟵ 3 4 5

the relevant information for X cannot be obtained until the right operand has been handled.

When an operator sub-expression is recognised, IDSTK (IOPTR) is tested to determine the operator. If a dyadic operator is present, the stack position IDSTK (IOPTR + 1) must also be tested. This address will have value 67 for an outer product (A∘.+ B is bracketed as (A∘+B) and ∘ is stacked before +). For an inner product, IDSTK (IOPTR + 1) will have value 71 .

This test enables inner and outer products to be handled by the same method.

Compound operations, such as +/X , present no difficulty as the symbol '/' will be recognised immediately when IDSTK (IOPTR) is tested. The uses of the symbol '/' (also '⌿') for reduction and compression have already been distinguished during the lexical scanning phase, and different macro numbers used for each.

4.3 Immediate Action Macros

A number of APL symbols require some immediate action when they are recognised during the left-to-right scan. These are:

(i) the symbols (and)

(ii) symbols used in indexed expressions viz. [,] and ;

(iii) the symbol ; used in heterogeneous output expressions

(iv) the symbols [and] used to specify a co-ordinate value .

4.3.1 The symbols (and)

Recognition of the symbol (during the left-to-right scan indicates the start of a new level of nesting. The stack pointers IDPTR, IDLPTR and IBIT (discussed previously) must be updated.

This action must be carried out immediately so that information for the new sub-expression (or function or subroutine call) can be stacked. The necessary action is produced by expansion of macro number 64.

Recognition of the symbol) during the left-to-right scan indicates that a complete sub-expression (or function or subroutine call) can now be dealt with. More information is stacked on IDSTK corresponding to the identifiers appearing within brackets. The relevant code can now be produced on the output stream. This process normally involves the expansion of a series of macros. Operand code is produced and stored temporarily in the array ITEMP. It is then obtained from ITEMP as required when an operator macro is being expanded.

IDSTK and IBITS are then unstacked as described previously and the left-to-right scan continues.

4.3.2 Symbols used in indexed expressions

Consider firstly the code produced corresponding to the APL statement

$$A \longleftarrow B[I\ ;\ J + 6] + 1$$

where A, I and J are scalars and B is non-scalar. The code is of the form shown below.

```
(i)    ZØ   =   Ø
(ii)   ZB1  =   ZPOINT (ZPT)
(iii)  ZPT  =   ZPT + 1
(iv)   ZINDX (ZB1 + 1)   =   I
(v)    ZINDX (ZB1 + 2)   =   J + 6
(vi)   ZPOINT (ZPT)      =   ZB1 + 2
```

(vii) CALL STARTS (B_{NAMES} ---)

(viii) CALL FIND1 (--- B_{NAMES} --- Y_n ---)

(ix) ZPT = ZPT +1

(x) Y_{n+1} = Y_n

(xi) IF(MARKER.NE.Ø) GOTO 1ØØ

(xii) A = Y_{n+1}

(xiii) GOTO 1Ø1

(xiv) 1ØØ CALL SPECS (A_{NAMES}, Y_{n+1}, ---)

(xv) 1Ø1 CONTINUE

Lines (x) to (xv) are required so that specifications of the following types may be handled correctly.

(a) scalar ← scalar expression

(b) vector ← non-scalar expression

(c) scalar ← non-scalar expression (implies type change of scalar)

These lines are obvious candidates for optimisation at a later stage, (see Chapter VIII, § 8.3) . Note that B_{index} is used to mean the index for B in NAMES. The lines of code have been numbered for ease of reference.

It can be seen that no reference is made to the non-scalar B in the code produced until after the subscripts have been handled and the symbol ']' obtained. However, the index for B is stacked on its recognition. In order to avoid confusion when dealing with the subscript expressions, therefore, a new base level is created on IDSTK when '[' is obtained. Lines (ii) and (iii) are also produced corresponding to the symbol '['. Some immediate action is thus required for the symbol '['.

The symbols ';' and ']' used in indexing serve as delimiters for the preceding subscripts. When either of the above symbols is recognised, therefore, the code corresponding to the subscripts can be produced. After the code for a subscript has been generated, the subscript information can then be unstacked. Any further subscript information obtained can be stacked in turn on IDSTK, starting from the base level set up when '[' was recognised.

Line (iv) is produced when ';' is obtained, while lines (v) to (ix) are produced when ']' is recognised. It can be seen that part of the action required for ';' and ']' is the same. Some additional action is required for ']' since it delimits not only a subscript expression, but also the complete indexed variable.

The code Y<integer> is stored on ITEMP to be used as a parameter for the + macro, the next to be expanded.

Return must be made to the previous base level when an indexed expression has been dealt with. The result is an intermediate expression and the value -1Ø is stacked to indicate this type of operand. However, in this case the value -1Ø should overwrite the non-scalar index in IDSTK.

Thus, all three symbols, '[', ';', ']' , are handled by expansion of immediate action macros. The action required for each is outlined below.

The symbol '['

1. Produce code of the form shown in lines (ii) and (iii) above.
2. Increment IBIT by 1

3. Increment IDPTR by 1

4. Set IDSTK (IDPTR) = IDLPTR

5. Set IDLPTR = IDPTR

The symbol ';'

1. Produce code of the form shown in line (iv) above

2. Set IDPTR = IDLPTR

The symbol ']'

1. As for step 1 for the symbol ';'

2. As for step 2 for the symbol ';'

3. Produce code of the form shown in lines (vi) to (ix) above

4. Set IBIT = IBIT - 1

5. Set IDPTR = IDLPTR - 1

6. Set IDSTK (IDPTR) = -1Ø

7. Set IDLPTR = IDSTK (IDLPTR)

The above illustrates the action required for a very simple array element reference. It should be noted that the sub-expression J + 6 would be bracketed and dealt with in the usual way. When the subscript J + 6 was handled on recognition of ']' , therefore, a -1Ø entry would appear on the stack. The code corresponding to J + 6 would thus be obtained from ITEMP.

The code produced corresponding to a non-scalar variable name is described in detail in Chapter V. It is obviously more complex, as every element of the non-scalar has to be accessed.

4.3.3 The symbol ';' used in heterogeneous output expressions

The constituents of a heterogeneous output statement are handled separately Each constituent is delimited by a semi-colon (or by a blank in the case of the last constituent).

An immediate action macro is expanded when ';' is encountered. (This use of ';' is distinguished from its use as a subscript separator in indexed expressions by the absence of enclosing square brackets.)

The macro produces a WRITE statement to write out the constituent of the heterogeneous output statement. A FORMAT statement is also produced. Constituents can be literal or numeric and two output statements are produced in each case. The FORMAT statements are such that all parts of a heterogeneous output statement appear on one line.

The stack pointers IDLPTR and IDPTR are both reset to Ø after handling a constituent of a heterogeneous output statement.

The variable IHET is set to 1 whenever a heterogeneous output statement is detected. Its value is tested at the end of a line. In this way the last constituent can be detected and handled correctly.

4.3.4 <u>The symbols '[' and ']' used to specify a co-ordinate value</u>

An example of the above use of '[' and ']' is in the statement

```
A ← + / [1] X
```

Again a new base level is created on IDSTK when '[' is recognised. This use of '[' is distinguished from its use in indexing by the presence of an operator immediately to the left of '[' . The variable ILEFT (mentioned previously) will have value Ø in this case.

ILEFT has to be updated within square brackets (so that nested co-ordinate specifications can be detected). Thus, ILEFT cannot be used to distinguish indexing and co-ordinate specifications when ']' is recognised.

For this purpose another variable, NCOORD, is used. It is increased by 1 whenever '[' is obtained in co-ordinate specifications and code of the form

```
          ZCDPTR   =   ZCDPTR + 1
```

is generated. The scan then continues in the normal manner until ']' is obtained. NCOORD is decreased by 1 and, if non-zero, macro number 19 is expanded to produce code of the form

```
          IF (ZCDPTR.GT.ZLIM14) CALL  GVOVER (14, & 1ØØ)
          ZCOORD (ZCDPTR)  =  <expression inside square brackets >
   1ØØ    CONTINUE
```

ZCOORD is a stack with pointer ZCDPTR in which successive co-ordinate values in an expression are stored. Before stacking another value in ZCOORD, a test is made for overflow. If the test is satisfied, GVOVER is invoked to print out a warning message.

(The value ZCOORD (ZCDPTR) is tested in the function FIND, used to handle the non-scalar reference. The required co-ordinate value can thus be obtained.)

A stack is required to handle nesting of co-ordinate specifications.

Now consider the example,

ϕ [1] A + ρ [2](X − Y + Z) + B

Then code of the form

```
                     :
          ZCDPTR   =   ZCDPTR + 1
                     :
          ZCOORD (ZCDPTR)  =  1
                     :
          ZCDPTR   =   ZCDPTR + 1
                     :
          ZCOORD (ZCDPTR)  =  2
                     :
```

The"scope" of [2] extends over X - Y + Z . After dealing with this expression, the value 2 must be unstacked from ZCOORD.

Action of this kind is organised by maintaining a stack of bracket counts. The count is increased for (and decreased for) . Thus, only when the matching) has been dealt with is ZCDPTR decreased. At this stage, code of the form

```
ZCOORD (ZCDPTR)  =  Ø
ZCDPTR  =  ZCDPTR - 1
```

is generated.

The stacks IDSTK and IBITS are then updated, and the scan continues.

4.4 Symbols Handled by Production of a 'FIND' Call with First Parameter Non-zero

The APL operators handled by producing a call of the function FIND are listed in Chapter I, § 1.2.5 . (For occurrences of these functions with (right) parameters not numeric non-scalars, similar techniques are applied using other functions. These functions are also listed in Chapter I.)

There are 14 APL operators in the above list. The first parameter of a FIND call has value Ø to 14, Ø indicating the normal accessing method, and 1 - 14 one of the methods for a specific operator.

The method of handling the above operators is as follows. The subscripts for the required element are set up in consecutive locations of the array ZINDX. A function is applied to these elements to produce the desired indices for the result.

Thus, for example, suppose the $(3,2)^{th}$ element of ϕA is required, where A is a 4x5 array. Then the result is obtained by accessing the $(3,5-2-1)^{th} \equiv (3,4)^{th}$ element of A . A simple function has been applied to the second subscript and then the normal accessing method (defined by FIND (∅,----) } is applied. The same principle is applied to the other operators in the list.

The right operand for one of the above operators can be an expression. All non-scalars in the expression would then have to be accessed in the manner determined by the operator. Thus, the first parameter value for the FIND call has to be retained throughout the scope of the operator. To allow for nesting of expressions involving the above operators, a stack of first parameter values, IFIND, is maintained. Consider, for example, the expression

$$\phi(\theta X) + Y$$

During the right-to-left scan, this expression would be bracketed as

$$(\phi((\theta X) + Y))$$

The following action is required during the left-to-right scan.

1. Recognition of (

    A new level is produced on IDSTK by updating IDLPTR and IDPTR, i.e. by setting:

    ```
    IDPTR        -    IDPTR + 1
    IDSTK (IDPTR)   -    IDLPTR
    IDLPTR       =    IDPTR
    ```

2. Recognition of ϕ

    ```
    Set  IFNPTR   -   IFNPTR + 1
    Set  IFIND (IFNPTR)   -  1
    ```

Set IOPTR = IOPTR - 1

Set IDSTK (IOPTR) = macro number for ϕ

3. Recognition of (

As for 1 above

4. Recognition of (

As for 1 above

5. Recognition of $\ominus$

Set IFNPTR = IFNPTR + 1

Set IFIND (IFNPTR) = 3

Set IOPTR = IOPTR - 1

Set IDSTK (IOPTR) = macro number for $\ominus$

6. Recognition of X

Stack the index for X on IDSTK

7. Recognition of)

Examine IDSTK (IOPTR) . Detect macro number for $\ominus$. Expand an operand macro to produce code for X . This involves a call of FIND having first parameter with value 3 (obtained from IFIND stack).

Now set IFNPTR = IFNPTR - 1 .

The scope of $\ominus$ is exceeded now and therefore the first parameter entry (value 3) can be removed. IFIND (IFNPTR) now has value 1. This is the correct value since the scope of ϕ has not yet been exceeded.

8. Recognition of +

Set IOPTR = IOPTR - 1

Set IDSTK (IOPTR) = macro number for +

9. Recognition of Y

Stack the index for Y on IDSTK

1Ø. Recognition of)

Expand operand macro twice to produce code for + operands and store the code in ITEMP. Then expand the + macro, which obtains the operands from ITEMP when required.

Expansion of the macro for the right operand results in production of a FIND call with first parameter value 1.

11. Recognition of)

The intermediate result code required for ⌽ has already been stored in ITEMP. The variable IFNPTR is then decreased by 1.

The resulting expression involving two YSTORE elements would be stored on ITEMP. IFIND (IFNPTR) retains its value over the entire range of an operator, that is, until the closing round bracket for the symbol has been dealt with.

The macro corresponding to the above operators only requires to unstack the top value from IFIND.

The user can specify that an operation of the above type is to be applied along the Jth co-ordinate, for example ⌽ [J] A .

The above method can still be applied by simply producing code of the form

```
        ZCDPTR   =   ZCDPTR + 1
        ZCOORD (ZCDPTR)   =   J
```

when [J] is recognised. This is done by a macro expansion. The value of ZCOORD (ZCDPTR) is tested in FIND to ensure that the operation is applied along the required co-ordinate.

There are a number of dyadic operators in the above group. These are handled in a similar way.

Consider the left-to-right scan for

$$(C \; \phi \; (A + B) \;)$$

where C, A and B are non-scalar.

When ϕ is recognised, the value 5 is stacked on IFIND. The index for C , the left operand of ϕ , must now be included in the relevant FIND calls.

Corresponding to the above expression, code of the following form would be generated.

⋮

Start of looping
instructions

⋮

<label > CALL FIND1 (5,1,C_{index},A_{index},--- Y < integer 1 > ---)

CALL FIND1 (5,1,C_{index},B_{index},--- Y < integer 2 > ---)

⋮

The code Y < integer 1 > + Y < integer 2> would then be placed on ITEMP.

A stack is maintained to allow for nesting of dyadic operators of the above group. This ensures that the correct NAMES index is inserted as third parameter of the FIND call.

The left operand of a dyadic operator of the above group need not be a numeric non-scalar. Thus a second stack, containing type values (to be used as second parameters in FIND calls) is also maintained.

Expressions are not allowed as left parameters for dyadic operators of the above group. This avoids the necessity for maintaining two sets of current bounds, one for the left and one for the right operand.

4.5 The Handling of □ and ⍞

The symbols □ and ⍞ used for output are very straightforward. For example, the statement

⍞ ← 'THIS IS AN EXAMPLE'

would be bracketed as

(⍞ 'THIS IS AN EXAMPLE').

□ and ⍞ used for output can thus be treated as any other monadic operators.

□ and ⍞ are slightly more complex when they appear in input expressions.

A temporary variable is introduced to store the values read in when □ or ⍞ is encountered. When □ or ⍞ used for input is met in the left-to-right scan, the values -2 and -1 respectively are stacked on IDSTK. When the next) is encountered and an operand macro is to be expanded, the value of IDSTK (IDPTR) is tested. If it is I = -1 or I = -2 , then macro number (I + 83) is expanded. This produces code to read in the required values.

Consider for example,

A ← , ⎕

This is bracketed in NCODE in the form

(A ← (,⎕)) .

The action required is outlined below.

1. Recognition of (

    Create a new base level on IDSTK

2. Recognition of A

    Stack the index for A in NAMES on IDSTK

3. Recognition of ←

    Set IOPTR = IOPTR - 1

    Set IDSTK (IOPTR) = macro number for ←

4. Recognition of (

    Create a new base level on IDSTK

5. Recognition of ,

    Set IFNPTR = IFNPTR + 1

    IFIND (IFNPTR) = 4

6. Recognition of ⎕

    Stack -2 on operand part of IDSTK

7. Recognition of)

Expand macro number 81 to produce code for the input operation required. Store the result variable in ITEMP and return to the previous level of IFIND. (The values read in are stored and treated as a vector by applying the function IRFIND (FIND for intermediate results).)

The stacks IDSTK and IBITS are then updated.

8. Recognition of)

The analysis now proceeds in the usual way.

It is essential always to test for the symbols ⎕ and ⍞ used in place of operands, and expand the appropriate macro if the test is satisfied.

4.6 An Example Showing the Process Carried Out During the Left-to-Right Scan

Consider the left-to-right scan applied to

A ←— B [C + D * E] FN X

The reader is referred to Diagrams 3.4(a) and 3.4(b) showing the states of the arrays NAMES, NOLINE and NCODE corresponding to the above statement. B is non-scalar, FN is a function name and all the other variables are scalar.

At the start of the left-to-right scan, the variables IDLPTR, IDPTR and IBIT are zero, and IOPTR has value 5Ø1 . IDSTK is set to zeros and the elements of IBITS are false. The action carried out is described below.

1. Recognition of -64 (macro number for (is 64)

```
Set  IDPTR  =  IDPTR + 1
Set  IDSTK (IDPTR)  =  IDLPTR
Set  IDLPTR  =  IDPTR
Set  IBIT  =  IBIT + 1
```

2. Recognition of index for A

```
Set  IDPTR  =  IDPTR + 1
Set  IDSTK (IDPTR)  =  index for A
```

3. Recognition of -3 (macro number for ← is 3)

```
Set  IOPTR  =  IOPTR - 1
Set  IDSTK (IOPTR)  =  3
Set  IBITS (IBIT)  =  .TRUE.
```

4. Recognition of -64

As for step 1 above

5. Recognition of index for B

```
Set  IDPTR  =  IDPTR + 1
Set  IDSTK (IDPTR)  =  index for B
```

6. Recognition of -62 (macro number for [is 62)

Produce 2 lines of code as described in §4.3 . Also create a new base level in IDSTK as described in step 1 above.

7. Recognition of -64

As for step 1 above.

8. Recognition of index for C

Set IDPTR = IDPTR + 1

Set IDSTK (IDPTR) = index for C

9. Recognition of -21 (macro number for dyadic + is 21)

Set IOPTR = IOPTR - 1

Set IDSTK (IOPTR) = 21

Set IBITS (IBIT) = .TRUE.

10. Recognition of -64

As for step 1 above

11. Recognition of index for D

Set IDPTR = IDPTR + 1

Set IDSTK (IDPTR) = index for D

12. Recognition of -25 (macro number for dyadic * is 25)

Set IOPTR = IOPTR - 1

Set IDSTK (IOPTR) = 25

Set IBITS (IBIT) = .TRUE.

13. Recognition of index for E

Set IDPTR = IDPTR + 1

Set IDSTK (IDPTR) = index for E

14. Recognition of -65 (macro number for) is 65)

Test IBITS (IBIT). This has value .TRUE., indicating a sub-expression. Test IDSTK (IOPTR). This has value 25, a dyadic operator macro number. IDSTK (IOPTR + 1) has value 21, and thus an inner or outer product has not been detected.

Now handle the operands for * . Produce more information on IDSTK for D as described in §4.1 . Then expand operand macro and store the code for D. Repeat the above process for E.

Now expand the * macro and produce the code (D**E), which is placed on ITEMP (the entries for D and E on ITEMP are removed).

```
Then   Set   IBIT    =   IBIT - 1
       Set   IDPTR   =   IDLPTR
       Set   IDLPTR  =  IDSTK (IDLPTR)
       Set   IDSTK (IDPTR)   =   -1Ø
       Set   IOPTR   =   IOPTR + 1
```

15. Recognition of -65

Using a similar process as for step 14 above, the code (C + (D**E)) is stored in ITEMP.

16. Recognition of -63 (macro number for] is 63)

Produce code as described in §4.3 .

```
Then   Set   IBIT    =   IBIT - 1
       Set   IDPTR   =   IDLPTR - 1
       Set   IDLPTR  =  IDSTK (IDLPTR)
       Set   IDSTK (IDPTR)   =   -1Ø
```

17. Recognition of index for FN

```
Set   IDPTR  =  IDPTR + 1
Set   IDSTK (IDPTR)  =  index for  FN
```

18. Recognition of index for X

```
Set   IDPTR   =   IDPTR + 1
Set   IDSTK (IDPTR)   =   index for X
```

19. Recognition of -65

Test IBITS (IBIT). This has value .FALSE., indicating a function or subroutine call. More operand information is set up on IDSTK and an operand macro is expanded to handle the function or subroutine call.

```
Then  Set   IBIT    =   IBIT - 1
      Set   IDPTR   =   IDLPTR
      Set   IDLPTR  =   IDSTK (IDLPTR)
      Set   IDSTK (IDPTR)   =   -1Ø
```

2Ø. Recognition of -65

Carry out a similar process as for step 14 above.

Note that, for convenience, two consecutive entries of -65 have twice been used in place of an entry of +162. This is simply for ease of explanation.

Also, in describing the processes carried out, test for overlap of the stack pointers IDPTR and IOPTR have been omitted. Similarly, an overflow test for IBITS has been omitted.

CHAPTER V

THE MACRO METHOD

This chapter describes the method of producing target-language code using macros. A complete list of macro instructions and their functions is given in Appendix 3.

Macro bodies are stored on disc. At the time of a macro expansion, all the necessary parameter information has been stacked on IDSTK, as described in Chapter IV.

The start address for a macro body is obtained from the table MCADDR. To expand macro number N, for example, the start address is given by MCADDR (N). Macro bodies are in card image form and the first line, IV, of any macro body is given by

IV = <start address> /8∅ + 1

5∅ records, starting from the IVth, are then read into an array MACROS. The first position to be accessed within the starting record is given by

IP = <start address> - (IV-1)*8∅

Thereafter, each character in turn of the macro body is accessed until the end of the macro body is reached. Access is sequential within a macro body unless altered by use of branching instructions. Such instructions are described in §5.1.6 . Instructions within a macro body are separated by two blank characters. All components of a macro instruction are separated by 1 blank character and labelled instructions have 1 blank between : and the corresponding instruction, (see §5.1.6) .

Macros are the means by which target language code is accumulated on the array MTEMP until ready to be transferred to the output medium. The contents of MTEMP will be transferred to the output medium

(i) when a complete line of code has been produced

(ii) when a character is to be stored on MTEMP and the pointer TEMPR has value 73. (lines of FORTRAN code do not exceed 72 characters).

For case (ii) above, a continuation line is produced and the process is repeated until the line is complete.

There is continual interchange between the arrays ITEMP and MTEMP. The function of ITEMP has been discussed in Chapter IV. It is a temporary storage place for operands. The method of transfer between MTEMP and ITEMP is discussed in §5.1.2 . Transfer of information from IDSTK to MTEMP is described in §5.1.5 . §5.1 categorises the macro instructions into a number of different groups.

5.1 Groups of Macro Instructions

All the macro instructions defined fall into one of the groups listed below.

1. Instructions which access MTEMP.
2. Instructions which transfer information between MTEMP and ITEMP.
3. Instructions which produce lines of code on the output stream.
4. Instructions which produce code on MTEMP.
5. Instructions which transfer information from IDSTK to MTEMP.
6. Branching instructions.
7. Terminating instructions.
8. Looping instructions.
9. Instructions which update pointers.
10. Instructions which set the values of global variables.
11. Instructions to increment global variables.
12. Instructions to calculate expression values and store on MTEMP.

§5.1.1 to §5.1.12 describe each of the above groups in more detail.

5.1.1 Instructions which access MTEMP

Target language code can be placed on MTEMP using the macro instruction

% --- TEXT --- %

This instruction inserts the string --- TEXT --- on MTEMP, starting from the current position of MTEMP. The pointer TEMPR for the array MTEMP is updated as required during the code production stage.

The macro instruction

&

transfers the contents of MTEMP to the output medium. This instruction is used when a complete line of code has been accumulated on MTEMP. In addition, there are a number of composite macro instructions. These produce lines of code (first accumulated on MTEMP) on the output medium. Examples are given in §5.1.3 .

5.1.2 Instructions which transfer information between MTEMP and ITEMP

Let us suppose that an operand macro has been expanded to produce code for an operand. This is accumulated on MTEMP. When complete, use of the macro instruction

S

causes the code to be transferred from MTEMP to ITEMP. This is done in the following manner. The contents of MTEMP(7) to MTEMP(TEMPR) are transferred to locations (IBPTR+1) to (IBPTR + TEMPR-6) of ITEMP. ITEMP(IBPTR+TEMPR-5) is then set to IBPTR, and IBPTR is updated to (IBPTR+TEMPR-5) .

Thus, for example, suppose TEMPR is 1Ø, IBPTR is 1 and MTEMP(7) to MTEMP(1Ø) contain the characters ABCD . After using the S instruction, the contents of ITEMP are

Ø,A,B,C,D,1 - - - - -

and IBPTR is 6.

In addition, MTEMP is reset to blank characters and TEMPR is set to 7, the starting position for most lines of FORTRAN code.
The pointer TEMPR is automatically reset to 7 after clearing MTEMP. This can be over-ruled using the macro instruction,

T <integer>

where <integer> can be any positive integer i such that 1 ≤ i ≤ 8Ø. This instruction is described in §5.1.11 .

The macro instruction S+ is similar to the S instruction, except that MTEMP is not cleared after the transfer.

The operand macros (referred to above) are described in detail in §5.2 .

Now let us suppose that the two operands ABCD and XYZ for a dyadic operation have been stored on ITEMP by the above method. Suppose that ITEMP has been set up as shown in Diagram 5.1.2(a), and that IBPTR has value 1Ø. Then the left operand, ABCD, can be reproduced on MTEMP, when required in an operator macro body, by use of the macro instruction

LO

This instruction transfers the contents of ITEMP (IX+1) to ITEMP (IY-1) to MTEMP, starting from position TEMPR. IX and IY are given by

IY = ITEMP (IBPTR)

IX = ITEMP (IY)

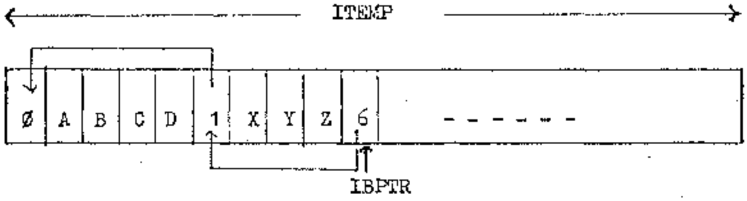


Diagram 5.1.2(a) : Shows a possible structure for the array ITEMP .

The contents of ITEMP are unaltered by this instruction. However, a marker, NLEFT, is set to 1 to indicate that two entries are to be removed from ITEMP after the right operand, XYZ, is accessed. If the entries had been placed in ITEMP in reverse order (and thus IREV is set to 1) then the variable NLEFT is set to 2. In this case no entries are to be removed from ITEMP. The variable NLEFT is tested when the macro instruction RO (see below) is handled. The number of entries to be removed from ITEMP is thus determined.

Thus, if ITEMP has the structure shown in Diagram 5.1.2(a) and the macro instruction LO is executed, then MTEMP will have the structure shown in Diagram 5.1.2(b) , assuming MTEMP has just been cleared.

The right operand for a dyadic operator can be transferred from ITEMP to MTEMP by use of the macro instruction

RO

This instruction transfers the contents of ITEMP (IY+1) to ITEMP (IBPTR-1) to MTEMP, starting from position TEMPR. IY is as defined previously.

NLEFT is tested to determine the number of entries to be removed from ITEMP. Thus,

if NLEFT is Ø , 1 entry is removed
if NLEFT is 1 , 2 entries are removed
if NLEFT is 2 , Ø entries are removed.

Thus, if the macro instruction RO is now used, the structure of MTEMP would be as given by Diagram 5.1.2(c) . After using this combination of instructions, ITEMP would be empty and IBPTR would have value 1.

The instruction RO+ is similar to RO , except that ITEMP remains unaltered by the instruction.

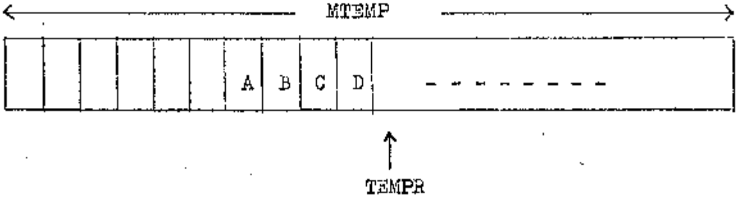


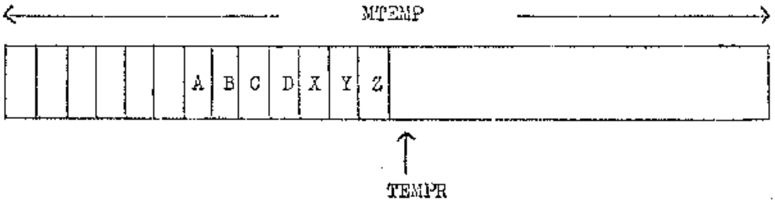
Diagram 5.1.2(b) : Shows the structure of MTEMP obtained by using the LO instruction for an ITEMP configuration as shown in Diagram 5.1.2(a) .

A third macro instruction R1 may be used. This instruction transfers the right-most entry of ITEMP to MTEMP, but first removes any enclosing round brackets. Similarly, the instruction R1+ is defined.

The composite instructions PL and PL+ are also defined. These produce code of the form

Y<integer> = <left operand code>

where <integer> is any positive integer and <left operand code> is obtained from ITEMP.



The pointer TEMPR has value 14.

Diagram 5.1.2(c) : Shows the structure of MTEMP obtained by using the instructions LO and RO for an ITEMP configuration as shown in Diagram 5.1.2(a) .

NLEFT is set by use of PL, but is unaltered by use of PL+ .

Similar effects can be produced using the macro instruction sequences

```
          %Y <integer>  =%      LO
```

and

```
          %Y <integer>  =%      LO+
```

respectively (assuming MTEMP had just been cleared).

Similarly, the macro instructions PR and PR+ are equivalent to the instruction sequences

```
          %Y <integer>  =%      RO
```

and

```
          %Y <integer> = %      RO+
```

respectively, again assuming that MTEMP had been cleared previously.

5.1.3 Instructions which produce lines of code on the output stream

The majority of the macro instructions defined fall into this category. They are, in fact, composite instructions replacing groups of other instructions. Examples are:

(i) CS

This macro instruction produces a number of non-executable statements (for example, INTEGER, REAL, IMPLICIT, COMMON, EQUIVALENCE). They are generated after the code for a function header statement.

(ii) Z <integer>

This macro instruction produces code of the form

Z <integer 1> = <integer>

where <integer 1> is a positive integer and <integer> is any integer. This instruction is often used in conjunction with

(iii) Z+

The macro instruction Z+ produces code of the form

Z <integer 1> = Z <integer 1> + 1

where <integer 1> has been previously introduced by a statement of type (ii) above.

5.1.4 Instructions which produce code on MTEMP

These instructions are dependent on the current operator being handled, that is, on the current value of IDSTK (IOPTR). Examples are:

(i) RL

This results in production of one of the forms

.LT. .LE. .EQ. .GE. .GT. .NE.

on MTEMP, depending on whether IDSTK (IOPTR) has value 6,7,8,9,1Ø or 11 respectively.

(ii) AO

This results in production of

.AND. (a)

or

.OR. (b)

on MTEMP, depending on the value of IDSTK (IOPTR). For value 4, (a) is produced; for value 5, (b) is produced. These values correspond to the operators $\wedge$ and $\vee$ respectively.

5.1.5 Instructions which transfer information from IDSTK to MTEMP

Such instructions involve use of the ? symbol. For example,

? <expression>

transfers the value of IDSTK (IDLPTR + <expression>) to MTEMP. The value is placed starting from position TEMPR of MTEMP. This is the method of obtaining parameter information inside macro bodies.

<expression> is terminated by either a blank or a comma. It may contain arithmetic expressions whose operands are integers or any of the global variables listed below:

(i) IND - gives the current value of <integer> to be used in expressions of the form

Y <integer> = - - - - -

It is updated as described in §5.1.11 .

(ii) INE - gives the current value of <integer> to be used in expressions of the form

ZB <integer> = ZPOINT (ZPT)

(see Chapter IV).

(iii) S <integor> - gives the current value of SS(<integer>) . The function of array SS is described in § 5.1.8 .

Bracketing is allowed in the above expressions. Brackets may be nested up to a maximum of 1Ø levels deep.

The form ?? <expression> or ? (--- expression involving ? ---) may bc used, but ? may only be nested to two levels deep.

(A transition matrix is used to accumulate the expression value up to the terminating blank or comma. The value is then converted to character form and transferred to MTEMP.)

Valid examples are:

(i) ?4

This transfers the contents of IDSTK (IDLPTR + 4) to MTEMP.

(ii) ?(1 + (IND - 1) * 3)

This transfers the contents of IDSTK (IDLPTR + 1 + (IND - 1) * 3) to MTEMP.

5.1.6 Branching instructions

Branching macro instructions can be

(i) unconditional

(ii) conditional

For both (i) and (ii) there must be an associated labelled macro instruction. Labelled instructions take the form

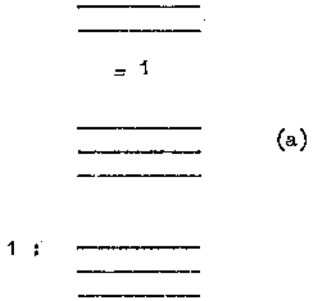
<label number>: <macro instruction>

<label number> is any positive integral value which is unique for a given macro body. The form < macro instruction > represents any valid macro instruction.

For case (i), there will be a corresponding statement of the form

= < label number >

This statement will cause a break in the sequential access of the macro bodies. The next instruction to be obeyed will then be <macro instruction>. For exemple, consider the following macro body:



When the instruction = 1 is reached, the subsequent instructions (a) will not be obeyed. The next instruction to be obeyed will be that labelled 1.

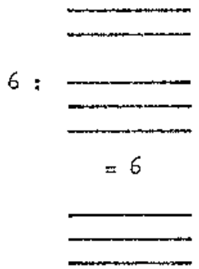
Branching forwards or backwards is handled in the following way. A 2-dimensional table, MLTAB, is maintained. For a particular row, the first entry gives a label number value and the second the pointer value (given by variable ICLPTR) for the array MACROS.

During the sequential scan of MACROS, if a labelled instruction is encountered, an entry is set up in MLTAB. When a branching instruction is met, for example

= n ,

MLTAB is scanned for an entry corresponding to n . If an entry exists, then a backward jump is made to the correct instruction (using the second part of the MLTAB entry).

Consider, for example,



When the instruction = 6 is encountered, MLTAB will have an entry of the form

6	n

where n is the value of ICLPTR corresponding to the blank after 6 : . A jump can thus be made to the correct point in the macro body.

If an entry does not exist in MLTAB, then a forward jump has been requested. When this occurs, MACROS is scanned sequentially for a labelled instruction. If a labelled instruction is met, an entry is set up in MLTAB. If the labelled instruction obtained is not the required one, the process is repeated until the correct instruction is found. Sequential execution of macro instructions is then resumed from the point reached.

MLTAB has 1ØØ rows and is accessed sequentially. This method allows nesting of label numbers to any depth. For example,

1 : 2 : < macro instruction>

would present no problem.

MLTAB is cleared on exit from each macro body.

A number of conditional branching instructions have been defined. In general these take the form

IF <g.v.n> <relational operator> m n

where (i) <g.v.n> is a global variable name

(ii) <relational operator> is one of EQ LT LE GE GT NE

(iii) m is a positive or negative integer

(iv) n is a positive integer.

Here the value of the global variable is compared with m. If the test is satisfied, a jump is made to the macro instruction labelled n. Otherwise, sequential execution continues.

Thus, for example, if IDSTK (IDLPTR + 1) has value 2 and the macro instruction

IF ?1 EQ 2 4

is executed, then a jump will be made to the macro instruction labelled 4. If, however, IDSTK (IDLPTR + 1) has value 3, no jump will be made and the instruction following the conditional branch will be executed next.

The allowable forms of the conditional branch instruction are given in Appendix 3.

5.1.7 Terminating instructions

The "unconditional stop" macro instruction is

#

Use of this instruction causes immediate exit from the macro body.

A number of "conditional stop" instructions have been defined. These take the form

IF <g.v.n.> <relational operator> m #

where <g.v.n>, <relational operator> and m have the same significance as in § 5.1.6 .

The allowable forms are again listed in Appendix 3.

An example is

IF ?3 LT 4 #

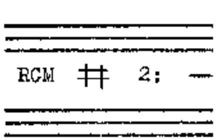
This instruction means "if parameter 3 is less than 4, stop. Otherwise, continue with sequential execution of macro instructions". (Parameter 3 is given by IDSTK (IDLPTR + 3) .)

An interesting use of the stop instruction is when preceded by RCM. The sequence

RCM #

causes execution of a macro to be interrupted while another macro is expanded. Return is afterwards made to the point in the original macro following # .

For example, suppose the macro below is being executed



when RCM # is reached, there is an immediate exit from the macro body. A second macro is expanded and then expansion of the above macro is resumed at label 2.

This facility was introduced to allow for the APL features of

(i) reduction

(ii) inner product

(iii) outer product

The handling of these features is described in detail in §5.3 and § 5.4 .

5.1.8 Looping instructions

The instruction

```
S<integer>,<expression>
```

is used in conjunction with the instruction

```
& < integer >
```

to produce looping.

Here < integer > can be from 1 to 1Ø and <expression> is subject to the rules laid down in § 5.1.5 .

The first instruction stores the value of < expression > (which is always integral for macro expressions) in SS(<integer>). SS is a 1Ø-element integer array. The current value of ICLPTR is also stored in variable ICOLM.

Sequential execution then continues until & < integer > is met.

This instruction tests the value, N, of SS (<integer>). If N $\geq$ 1, then SS (<integer>) is decreased by 1 and ICLPTR is reset to the value of ICOLM. If N < 1, then sequential execution of macro instructions is resumed.

This provides the facility of executing the same piece of macro code a variable number of times. For example, consider the following macro body

```
______________
______________
S1,3
______________
______________     (a)
______________
      & 1
______________
______________
```

Here macro instructions (a) will be executed 3 times.

5.1.9 Instructions which update stack pointers

A few instructions have been defined simply to update stack pointers. Examples are:

(i) STK

The effect of executing this instruction is:

(a) IDPTR is increased by 1

(b) a test is made for overflow of IDPTR and IOPTR. (If the test is satisfied, a message is printed out and execution is terminated.)

(c) IDLPTR is stored in IDSTK (IDPTR)

(d) IDLPTR is set to IDPTR.

This macro instruction is used to create a new base level on IDSTK.

(ii) RE

The effect of executing this macro instruction is to reset the value of IDPTR to IDLPTR. This instruction is used to reset IDPTR after expanding the macro for ; used in indexing.

5.1.1Ø Instructions which set the values of global variables

Two examples of macro instructions in this group are:

(i) MR <integer>

This macro instruction sets the value of the variable MARK to <integer>. The value can then be tested using a statement of the form

```
IF   MR   <relational operator>   m    n
```

(see §5.1.6) .

Thus, by setting the variable MARK, the path taken during expansion of a macro can be varied.

(ii) T <integer>

This macro instruction is used to set the value of the pointer TEMPR to <integer> . Usually, <integer> has value 1 to 6, since the most common use of the above instruction is to over-ride the setting of TEMPR to 7 after MTEMP has been cleared.

5.1.11 Instructions to increment global variables

The complete list of such instructions is given in Appendix 3. Examples are:

(i) +B

This increments the stack pointer IBIT by 1.

(ii) +D

This increments the variable IND by 1.

5.1.12 Instructions to calculate expression values and store on MTEMP

Besides the types of macro expressions mentioned previously, a number of others have been defined. The most widely used is

£< expression >

where <expression> is as defined in § 5.1.5 . £ gives the current value of IDOLR, the label value. (IDOLR is incremented as required to produce unique label numbers in the generated code by use of the macro instruction +I).

Suppose, for example, that IDOLR has value 1Ø4. Then £ - 4 will produce the label number 1ØØ on MTEMP.

5.2 The Use of Operand Macros

Two operand macros have been defined. Their functions are to use the information stacked in the operand part of IDSTK to produce the code for an operand and store it on ITEMP. The code for the operator macros is thus much simplified, as no type checking need be done in operator macros.

The first of the operand macros, referred to as the operand-A macro, merely determines the type of the operand being handled and produces the required code. It is discussed in detail in § 5.2.1 . Discussion of the second operand macro, referred to as the operand-B macro, is deferred until § 5.2.2 .

5.2.1 Operand-A macro

The operand-A macro is listed in Appendix 4. The macro body is explained below. It may be useful at this stage to recall the information stacked on IDSTK for each type of operand. The information is described in Chapter IV, § 4.1 .

The first requirement is to separate all operand types into groups which can (at least partially) be handled together. Thus, the macro body starts with a series of tests, the first being for an intermediate result operand. Such an operand is already present in ITEMP and thus no further action is required.

The functions of all macro instructions are listed in Appendix 3.

The test IF F EQ 1 1 is for a function or subroutine call. These are handled after label 1.

In each case, the final code produced on MTEMP is transferred to ITEMP before exit from the macro body.

If label 3 is reached on execution of the macro instructions, then the operand is either

(i) a scalar

or (ii) a niladic function name.

The code for the identifier is transferred from NAMES to MTEMP (using the instruction FN1), starting from position TEMPR.

Now consider the situation when label 2 is reached.

```
2: IF ?2 EQ Ø 3
```

This produces a branch to label 3 for a scalar identifier. Thus, if the above branch is not executed, the identifier types still to be distinguished are:

(i) literals

(ii) constant vectors

(iii) numeric non-scalars.

For each of these identifier types, loops are set up so that each element of the non-scalar may be accessed in turn. Only the first half of the loops is produced at this stage; the loops are not completed until either

(i) the end of the line is reached

or (ii) the dimensionality of the result changes.

The macro instruction SL produces code to start a loop, while the instruction FL generates the required code to end a loop. These are both defined in Appendix 3.

Non-scalar accesses result in generation of subroutine calls. Throughout this chapter, any subroutines referred to are present in the module library SARUN.

If the required loop-starts have already been produced by a previous expansion of the operand-A macro, there is no need to duplicate them. The variable MARKER will be non-zero if the loops have been started already. The instruction SL is composite and generates code of the form

```
          ZB <integer>  =  ZPOINT (ZPT)
          ZPT  =  ZPT + 1

          CALL STARTS (<operand index>, Z<integer1>, Z<integer 2>, ZNC )
                                  or
                                  ZF1
                                  or
                                  ZF2
          ZPOINT (ZPT)  =  ZB <integer> + Z <integer 2 >
          Z <integer 3 > =  1
<label 2 >Z <integer 4>  =  ZB <integer> + Z <integer 3>
          ZINDX (Z <integer 4 > )  =  1
          Z <integer 3 >  =  Z < integer 3> + 1
```

```
IF (Z <integer 3> .LE. Z <integer 2> ) GOTO <label 2>
Z <integer 5> = ZB <integer> + Z <integer 2>
Z <integer 6> = Z <integer 2> - 1
ZSAVE = Ø
```

Here < integer i > where $1 \leq i \leq 6$ are distinct positive integers. <integer> and < label 2 > are also positive integers.

The values <integer 5>, <integer 6 > and (<label 2> + 1) are stored. They will be used later when the loops are completed using the FL macro instruction. The code generated by the FL instruction is given in Appendix 3.

The subroutine STARTS has 1 input parameter I and 3 output parameters, J, K and L . I is the index of the non-scalar in NAMES.

The output parameters have the following significance:

(i) for a numeric non-scalar

J = the dope vector address

K = the number of dimensions

L = Ø if MARKER = Ø ; otherwise L = 1 .

(ii) for a constant vector

J = the number of elements

K = 1

L = Ø if MARKER = Ø ; otherwise L = -5 .

(iii) for a literal

J = the number of elements in the associated literal constant

K = the number of dimensions

L = Ø if MARKER = Ø ; otherwise L = -1 .

These parameters are used in subsequent subroutine calls.

At this stage, different subroutines are called for each type of operand. The possibilities are:

(i) FIND1 (containing a call of FIND) for numeric non-scalars

(ii) FIND2 (containing a call of UVFIND) for constant vectors

(iii) FIND3 (containing a call of LFIND) for literals.

Examples of tho code produced for each type of operand are given at the end of this subsection.

The 3 non-scalar cases are distinguished in the macro body. Thus,

```
IF   ?2   NE   1   4
```

separates the numeric non-scalar case from the others. A call of FIND1 is then generated.

The macro instruction FV provides the first 3 parameters for a FIND call (as described in Chapter I).

The instruction FX generates either

(i) the value of the NAMES index for the non-scalar

or (ii) ZF1

or (iii) ZF2

on MTEMP, depending on whether the identifier is a function or subroutine parameter (see Chapter IV).

%, Z% IND generates Z <integer 7> . This is an output parameter used to store the YSTORE index for the particular non-scalar element being accessed.

%, Y% IND generates an output parameter where the value of the non-scalar element is placed.

%, Z% IND-5 corresponds to parameter K of the STARTS call, while ZNC corresponds to parameter L .

The other parameters of FIND1 are global variables. The significance of all global variables is given in Appendix 8.

Finally, for numeric non-scalars, the value YSTORE (Y <integer 7>) is stored on ITEMP. Then follows a test for a constant vector identifier. This takes the form

```
        4:   IF   ?2   NE   -5   5
```

A call of FIND2 is then generated. FIND2 has the same parameters as FIND1 except that the global parameters ZCOORD and ZCDPTR are omitted. (These are unnecessary as they are used in specifying a co-ordinate value and constant vectors are one-dimensional.)

The value of the constant vector element being accessed is produced in Y < integer 7> .

For a literal identifier, a call of the function FIND3 is generated. A literal identifier is regarded as an array of elements whose values are the character values for the elements of the literal. FIND3 produces as a result the character value for the literal element in Z <integer 7>. The parameter Y <integer 7 > is unnecessary and has been omitted from the FIND3 parameter list.

The remaining case to be considered is that of function or subroutine calls. The macro instruction FA produces the entire code for the call, except for the list of global variables at the end. These variables, though not always used inside the function or subroutine body, must be inserted to allow for accesses of any global variables inside the body. (The user may, if he wishes, remove those found to be unnecessary on inspection of the generated code, or use the alternative method of handling global variables described in Chapter I, § 1.2.5 .)

EXAMPLES

Suppose that IBPTR is originally 1 and the operand-A macro is expanded in turn for the following identifiers:

(i) the scalar ABC

(ii) the function call A FN B , where A and B are both scalar

(iii) the numeric non-scalar subroutine parameter Y (Y is the left parameter of a subroutine)

(iv) the literal constant '1 3 4 5 9'

(v) the label name L1

(vi) the niladic function name F

(vii) the constant vector 3.1 2.4 6.7

Then, after the 7 macro expansions, ITEMP will have the structure shown in Diagram 5.2.1(a) .

No further code is generated for (i), (ii), (v) and (vi), but an entry is placed in LTABLE (see Chapter VI) for (v).

Assume that IND has value 1 and the label number value is 1ØØ. Assume also that INE has value 1Ø. Then corresponding to case (iii), the following code would be generated,

```
IF (MARKER.NE.Ø)  GOTO 1Ø1
ZB1Ø  =  ZPOINT (ZPT)
ZPT   =  ZPT + 1
CALL STARTS (ZF1, Z1, Z2, ZNC)
ZPOINT (ZPT)  =  ZB1Ø + Z2
Z3  =  1
```

1																	18
Ø	A	B	C	1	F	N	(	A	,	B	)	5	Y	3	T	O	R

19																	
E	(	Z	7	)	13	Z	7	24	1	Ø	3	27	F	31	Y	7	33

label value
associated
with L1
(see Chapter VI)

IBPTR

Diagram 5.2.1(a) : Shows the structure of ITEMP if IBPTR is initially 1 and the operand-A macro is expanded in turn for

(i) the scalar ABC

(ii) the function call A FN B , where A and B are both scalar

(iii) the numeric non-scalar left subroutine parameter Y

(iv) the literal constant '1 3 4 5 9'

(v) the label name L1

(vi) the niladic function name F

(vii) the constant vector 3.1 2.4 6.7

```
1Ø2     Z4  =  ZB1Ø + Z3
        ZINDX (Z4)  =  1
        Z3  =  Z3 + 1
        IF (Z3.LE.Z2)  GOTO 1Ø2
        Z5  =  ZB1Ø + Z2
        Z6  =  Z2 - 1
        ZSAVE  =  Ø
1Ø1     CALL FIND1 (Ø,Ø,Ø , ZF1,Z7,Y7,Z2,ZNC )

                    assuming
                    normal
                    access
```

Corresponding to case (iv), similar code to that produced above is generated, except that the last line is

```
1Ø1     CALL FIND3 (Ø,Ø,Ø , ZF1,Z7,Z2,ZNC )
```

Similarly, for case (vii), code of the above form is again generated. In this case, however, the last line has the form

```
1Ø1     CALL FIND2 (Ø,Ø,Ø , ZF1,Z7,Y7,Z2,ZNC )
```

5.2.2 Operand-B macro

The operand-B macro is expanded to handle the operands of certain dyadic mixed functions. These functions are:

(i) encode

(ii) decode

(iii) member

(iv) iota

(v) outer product

(vi) inner product

The method of handling these functions is discussed in § 5.3 .

A listing of the operand-B macro is given in Appendix 4.

The purpose of the operand-B macro is to store the elements of an operand in YROWL or YROWR. The arrays YROWL and YROWR are used to store the elements of the left and right operand respectively. YROWL has pointer ZROWNO, which is set to the number of elements in the left operand. Similarly YROWR has pointer ZROWNA.

Certain other functions are performed by the operand-B macro. These are determined by the value of the variable ZMARK. There are 6 possible values of ZMARK (Ø to 5) and the corresponding functions are listed below.

VALUE OF ZMARK	FUNCTION OF OPERAND-B MACRO
Ø	Store the N elements of the operand in YROWL and set ZROWNO = N
1	As for ZMARK = Ø . In addition, the current bounds, ZCBNDS, and pointer, ZCPTR, are updated to the bounds for the operand.
2	Store the N elements of the operand in YROWR and set ZROWNA = N . Update ZCBNDS and ZCPTR as described for ZMARK = 1 . Generate a call of the function INDX. This function is used to produce the result for the dyadic iota function (see § 5.3).
3	Store the N elements for the operand in YROWR and set ZROWNA = N .
4	As for ZMARK = 3 . In addition, the current bounds are updated as required for an inner product. This is discussed more fully in § 5.3 .

VALUE OF ZMARK	FUNCTION OF OPERAND-B MACRO
5	As for ZMARK = 3 . In addition, the current bounds are updated as required for an outer product. This is discussed more fully in § 5.3 .

The operand-B macro carries out its functions by generating a number of subroutine calls. The individual subroutines are present in the module library SARUM .

5.3 Handling of Mixed Functions

A number of mixed functions are handled by generating FIND calls. These were discussed in Chapter I.

This section describes the handling of

(i) functions involving expansion of operand-B macro

(ii) monadic rho

(iii) dyadic rho

(iv) monadic iota

(v) grade-up and grade-down

(vi) deal (dyadic ?)

5.3.1 Functions involving expansion of operand-B macro

The encode function

R ⊤ N

where R is a vector and N is a scalar, is handled in the following way.

The elements of R are stored in the array YROWL by expanding the operand-B macro with ZMARK set to 1. The dimension of R ⊤ N is the same as the dimension of R .

The scalar N is then produced on ITEMP by expanding the operand-A macro.

The operands having been dealt with, the encode macro (number 13) is then expanded. A call of the subroutine NCOAD is generated, that is

```
        CALL  NCOAD(N)
```

The result of R ⊤ N is a vector. It is stored in the array ZTEMP from the base level onwards. A series of non-scalar result elements may be stored in ZTEMP and hence a stack of successive base levels is required. The stack is represented by ZY and has stack pointer ZYPTR.

Non-scalar integer results are stored in ZTEMP; non-scalar real results in the array YTEMP. There is a similar stack ZYY, having stack pointer ZYYPTR, which gives successive base levels of YTEMP.

A call of the function NCOAD is generated to produce the required result for R ⊤ N and store it in the next level of ZTEMP.

The decode function

```
        R  ⊥  X
```

where R and X can both be non-scalar, also belongs to this group . The result of R ⊥ X is a scalar. Scalar arguments are extended to the same size as the other argument. For example, both

```
        10  10  10  10 ⊥ 1  7  7  6
  and               10 ⊥ 1  7  7  6
  is        1776 .
```

Taking this feature into account, the elements of the left operand are stored in YROWL by expanding the operand-B macro with ZMARK set to ∅ . The elements of the right operand are then stored in YROWR by expanding the operand-B macro with ZMARK set to 3 .

To produce the desired effect, the decode macro (number 14) simply generates a call of the subroutine DCODE. DCODE accesses YROWL and YROWR to produce the result N. The parameters for DCODE are N, YROWL, YROWR and ZCPTR.

The member function

A ∈ B

is another function belonging to this group. A and B can be non-scalar and A ∈ B has the same dimensions as A .

The elements of the left operand are stored in YROWL by expanding the operand-B macro with ZMARK set to 1. The right operand elements are stored in YROWR by expanding the operand-B macro with ZMARK set to 3. The result will thus have the same bounds as the left argument.

The member macro (number 12) simply generates a call of the subroutine MEMBAR to produce the desired result in ZTEMP.

The iota function,

A ⍳ B

where A and B may both be non-scalar, is very simply handled by this method. To produce the required result, in ZTEMP, it is only necessary to do two expansions of the operand-B macro.

The first expansion, with ZMARK set to ∅, stores the elements of the left operand in YROWL. The second expansion, with ZMARK set to 2, stores the elements of the right operand in YROWR. In addition, it sets up bounds (the result having the same bounds as the right operand) and generates calls of the function INDX. There is one call of INDX for each element of the right operand. INDX has parameters YROWL, YROWR and Z, where Z gives the position in YROWL of the right operand element being considered. The parameters are used to determine the result and store it in successive locations of ZTEMP, starting from the current base level.

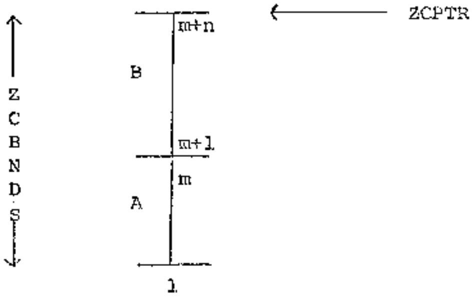
The outer product function is interesting. The non-scalar operands, A and B, of

A ∘ . f B

where f is any scalar dyadic function, are handled as follows.

First the left operand elements are stored in YROWL by expanding the operand-B macro with ZMARK set to 1. Bounds are also produced in ZCBNDS as for the left operand A .

The operand-B macro is then expanded with ZMARK set to 5. The elements of B are thus stored in YROWR. In addition, ZCBNDS is updated so that the bounds for B are stored above the bounds for A . Thus, if A is m-dimensional and B is n-dimensional, ZCBNDS will have (m+n) elements and ZCPTR will be set to m+n .



The outer product macro is then expanded. Since the dyadic function, f, can vary, the execution of the outer product macro (number 67) is interrupted at a certain point and the macro for f is expanded. The appropriate elements of YROWL and YROWR are first stored on ITEMP to be used by the f macro. Expansion of the outer product macro is then resumed from immediately after the first exit point.

This can be better understood by considering the code generated. This code is listed below. (The subroutine call

```
        CALL BDNO (Z, Z1 )
```

produces in Z the product of the first Z1 elements of ZCBNDS.)

```
        CALL  BDNO (Z1, ZROWNA )
        CALL  BDNO (Z2, ZROWNO )
          Z3  =  ZYY (ZYYPTR)
          Z4  =  Ø
          Z5  =  Ø
   1ØØ    Z5  =  Z5 + 1
          Z6  =  Ø
   1Ø1    Z6  =  Z6 + 1
          Z4  =  Z4 + 1
```

At this stage, the entries YROWL (Z5) and YROWR (Z6) are stored on ITEMP. Execution of the outer product macro is then interrupted using the macro instruction sequence RCM #.

The operands for the f macro have now been stored in ITEMP. The f macro is expanded to produce a result in ITEMP. For example, if f represents + , then the code (YROWL (Z5) + YROWR (Z6)) is stored on ITEMP in place of YROWL (Z5) and YROWR (Z6) .

Expansion of the outer product macro is resumed and the following code is generated.

```
          YTEMP (Z4 + Z3)  =  <result code for  f  macro with brackets
                                                         removed>
          IF (Z6.LT.Z1)  GOTO 1Ø1
          IF (Z5.LT.Z2)  GOTO 1ØØ
          MARKER  =  -5
          Z7  =  Ø
1Ø2       Z7  =  Z7 + 1
```

The code YTEMP (Z7 + Z3) is stored in ITEMP to be used as an operand in the next macro expansion.

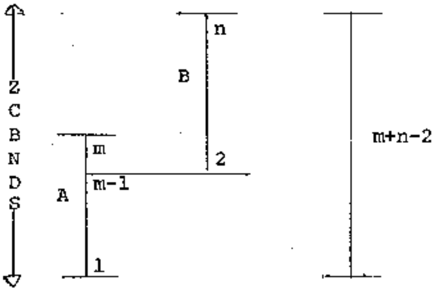
The inner product function

A f . g B

is handled by a similar method. Here A and B may be non-scalar and f and g are any scalar dyadic functions.

The operands A and B are handled as follows. The elements of A are stored in YROWL by expanding the operand-B macro with ZMARK set to 1. The bounds for A are also set up in ZCBNDS. The operand-B macro is then expanded with ZMARK set to 4. The elements of B are thus stored in the array YROWR. ZCBNDS is updated as required for an inner product. Thus, if A is m-dimensional and B is n-dimensional, ZCBNDS will be as illustrated below.

The inner product macro employs a similar strategy to that used in the outer product macro. In this case there are two scalar dyadic functions to be handled. There are thus two interruptions in the execution of the inner product macro. The inner product macro is listed in Appendix 4.



ZCBNDS (m)
is stored in
ZROWDM to be
used in the
inner product macro body.

ZCPTR has value m+n-2

5.3.2 Handling of the monadic RHO operator

This simply involves updating the contents of the array ZCBNDS.

For non-scalar result operands, the effect is produced by generating a call of the subroutine MRHO. For other non-scalar operands, the effect is produced by generating a call of the subroutine NRHO.

MHRO and NRHO are contained in the module library SARUN.

5.3.3 Handling of the dyadic RHO operator

The dyadic rho operator is handled by two macro expansions. The first generates code to store the left operand elements in the array ZROW. The second macro expansion generates a number of subroutine calls. These subroutines access the elements of ZROW (and the right operand) to produce the required result.

The dyadic rho operator is discussed in § 5.5 .

5.3.4 Handling of the monadic iota operator

To handle

⍳ N

where N is an integer ≥ ∅, code is simply generated to store values 1,2,...,N in successive positions in the array ZTEMP, starting from the current base level.

The variables, MARKER, ZCENDS and ZCPTR are set accordingly.

5.3.5 Handling of the grade-up and grade-down functions

To handle, for example,

⍋ A

where A is non-scalar, the result cannot be determined until all the elements of A are known. Expansion of the operand-A macro to handle A simply produces the start of the loops required for non-scalar accessing. Code is therefore generated to store the elements of A in the array YGRAD and the loops are then completed.

A call of the subroutine GRAD is then generated to produce the required result. The Quick-sort method of sorting (Knuth[2]) is used in GRAD to handle grade-up and grade-down.

5.3.6 Handling of the dyadic ? function

The operands A and B of

A ? B

where A and B are both scalar, are handled by two expansions of the operand-A macro. The code for A and B is thus placed in ITEMP. Then A calls of the function QUERY2 are generated. QUERY2 has parameters B and the array ZBOOL. A random number, N, in the range 1-B is first generated. ZBOOL (N) is then set. If, in a subsequent call of QUERY2, N is obtained as result, the previous occurrence of N can be detected by testing the value of ZBOOL (N) . If ZBOOL (N) is set, then the process is repeated until a random number is produced which has not occurred before. In this way, for example, N ? N will produce a permutation of N as result.

After A calls of QUERY2, ZBOOL must be reset in preparation for any subsequent series of calls.

5.4 Other Interesting Functions

This section deals with the operators

(i) specification ←

(ii) reduction f/ where f is any scalar dyadic function.

5.4.1 Handling of specification statements

For most functions, the left operand is handled before the right operand. The specification function is an exception. For specification, the right operand must be examined first so that the correct type and dimension information can be set up for the left operand. For example, consider

A ← L

If L is literal, then A will be literal also.
If L is numeric, then A will be numeric also.
In addition, the dope vector entry for A depends on the dope vector entry for L .

Thus, after handling the right operand, the appropriate information is stacked for the left operand. The specification macro generates a subroutine call, the subroutine being determined by the type information available.

After handling a specification operation, the index for the left operand is stacked in the appropriate location of IDSTK. The index is thus available to be re-used to provide operand information for the next operator in multiple specification statements. When the left operand is not an expression, no result code is placed in ITEMP.

However, consider

A ←— B + R [1] ←— 3 .

After handling R[1] ←— 3, a -1∅ entry is stacked on IDSTK. This is consistent with the above method, as the "index" for the left operand R [1] is -1∅, since R [1] is a result. Since -1∅ will now be used as the information for the right operand of +, it is necessary to have the code corresponding to R [1] stored on ITEMP.

5.4.2 Handling of reduction

Consider

f / X

where f is any scalar dyadic function and X is non-scalar. (The method employed only requires explanation for the non-scalar case.)

Assume, for ease of explanation, that X is a vector of N elements. Then the result required is

X [1] f X [2] f --- f X [N] .

The method employed is to set the variables OPL and OPR initially to

(i) the identity element, ID, for function f , and
(ii) X [N]

respectively. OPL and OPR are then stored in ITEMP.

Interruption of the reduction macro is then effected using the macro instructions RCM # discussed previously.

The f macro is then expanded to produce the result (OPR f OPL) on ITEMP. Expansion of the reduction macro is then resumed.

OPL is now set to the result on ITEMP, and OPR to X [N-1] . The whole process is repeated using the new values of OPL and OPR .

These steps are carried out N times, the result required being accumulated in OPL .

The reduction macro is listed in Appendix 4. For a non-scalar operand, code of the form listed in Chapter VIII, Example 8.1.9, is generated.

5.5 Examples of Macro Bodies

This chapter ends with three example macro bodies and their explanations. The examples chosen are:

(i) the macro for +, -, x, /, *

(ii) the macros for dyadic rho

(iii) the macro for quad input.

5.5.1 The macro for +, -, x, /, *

This macro is reproduced below.

```
BR   LO   O   RO   CB   S
```

The function of BR is simply to produce (on MTEMP. The left operand code is then transferred from ITEMP to MTEMP. (The operand code has previously been produced on ITEMP by two expansions of the operand-A macro.)

The instruction O produces the code +,-,*,/ or ** on MTEMP, depending on the operator being handled.

RO results in the transfer of the right operand code from ITEMP to MTEMP. CB produces a closing round bracket) on MTEMP.

Consider, for example,

```
A * B
```

where A and B are both scalar. Then, at this stage in the macro expansion MTEMP will contain (A ** B).

The contents of MTEMP are now transferred to ITEMP using the S macro instruction and exit from the macro body is effected using #.

Now consider

```
C + A * B .
```

Suppose A * B has been handled as described above. ITEMP now contains an entry (A ** B). The code for C is stored on ITEMP by expanding the operand-A macro. The variable IREV is set to 1 to indicate that the operands have been stored in ITEMP in reverse order. IREV is tested when LO and RO are used to determine the order of access of the operands.

The + macro is now expanded to produce (C + (A**B)) on ITEMP. Brackets are inserted round the result code each time the above macro is expanded. It was thus unnecessary to bracket A**B when RO was executed. It is, however, necessary to remove the outer brackets when an expression such as C + A*B is used to the right of a specification arrow.

5.5.2 The macros for dyadic RHO

As stated previously, dyadic rho is handled by two macro expansions.

These macros are listed in Appendix 4.

The first macro handles the left operand, A, of A ρ B .

Three operand types are first distinguished. These are:

(i) scalar

(ii) non-scalar or constant vector

(iii) result.

The first test IF ?1 LT Ø 1 distinguishes case (iii) above. Thus the code from label 1 onwards handles a result left operand.

The second test IF ?2 NE Ø 2 distinguishes case (i) above. Thus case (i) is handled up to label 2 and then case (ii) up to label 1.

For case (i), the following code is generated:

```
ZROW(1)  =  <scalar identifier>
ZROWNO   =   1
```

For case (ii), the following code is generated:

```
CALL RHOBND (<NAMES index>,<type value>) .
```

In RHOBND, the operand elements are stored in ZROW and ZROWNO is set.

For case (iii), scalar and non-scalar results are distinguished by testing the value of MARKER. The following code is generated:

```
IF (MARKER.NE.Ø)  GOTO <label 1>
ZROW(1)  =  <result code>
ZROWNO   =  1
GOTO <label 2>
```

```
<label 1>  Y <integer> = <result code>
           IF (Z <integer 1>.GT.ZLIM2Ø) CALL GVOVER (2Ø, &<label 2>)
           ZROW (Z <integer 1>)  =  Y <integer>
           CALL BDNO (Z <integer 2>,ZCPTR )
           IF (Z <integer 1>.LT.Z <integer 2>) GOTO <label 1>
           ZROWNO  =  Z <integer 2>
<label 2>  CONTINUE
```

Here ZLIM2Ø is the upper bound for ZROW. If it is exceeded, GVOVER is called to print out an error message. GVOVER handles overflow of all global non-scalars.

The call of BDNO produces the product of the ZCPTR elements of ZCBNDS in Z <integer 2>. This provides a terminating condition for the loop.

The right operand can now be handled. More operand types are allowable here. Thus the second macro has a greater number of tests. For each operand type, a different subroutine call is generated. These subroutines produce the required result in each case.

The following operand types are distinguished:

(i) scalar
(ii) literal
(iii) empty vector
(iv) numeric non-scalar
(v) constant vector
(vi) result

For each operand type, a call of the subroutine YRHO is generated. This produces in the first parameter the product of all the elements of ZROW, from positions 1 to ZROWNO. Thus, for the six cases listed above, the following code is generated:

(i) Scalar

```
            CALL YRHO (Z <integer 1>,ZROW)
            Y <integer 2> = <scalar identifier>
            CALL RHO2 (Y <integer 2>,Z <integer 1>)
            Z <integer 3 > = ZYY (ZYYPTR)     produced by instruction ZS
            Z <integer 4 > = Ø                } produced by
<label 1 > Z <integer 4> = Z< integer 4> + 1  } instruction DS
```

The values < integer 4 > and<label 1> are stored to be used in finishing the loops later using the PC macro instruction. The code

```
            YTEMP (Z < integer 4 > + Z < integer 3 > )
```

is stored in ITEMP to be used as an operand in the next macro expansion.

(ii) Literal

```
            CALL YRHO (Z < integer 1>)
            CALL RHO5 (<NAMES index>,Z <integer 1>)
            Z <integer 3> = ZY (ZYPTR)
            Z <integer 4> = Ø
<label 1 > Z <integer 4 > = Z <integer 4>+ 1
                        .
                        .
                        .
```

An entry ZTEMP (Z <integer 4 >+ Z <integer 3>) is again placed in ITEMP.

(iii) Empty vector

```
            CALL YRHO (Z < integer 1>)
            MARKER = -3
            CALL RHO4
```

A dummy entry is stored in ITEMP for an empty vector result. It is accessed during subsequent macro expansions (this was done for generality) but is detected by the fact that MARKER has value -3 .

(iv) Numeric non-scalar

```
            CALL YRHO (Z<integer 1>)
            CALL VECASE (<NAMES index>)
                                  or
                                 ZF1
                                  or
                                 ZF2
            CALL RHO4
```

An entry as for case (i) is placed in ITEMP.

(v) Constant vector

```
            CALL YRHO (Z<integer 1>)
            CALL RHO1 (<NAMES index>,Z<integer 1>)
            Z<integer 3> =  ZY (ZYPTR)
            Z<integer 4> =  Ø
<label 1>   Z<integer 4> =  Z<integer 4>+ 1
```

Again, an entry as for case (i) is placed on ITEMP.

(vi) Result

```
            CALL YRHO (Z<integer 1>)
            Y<integer 2> = <result code>
            IF (MARKER .NE. Ø)  GOTO<label 1>
            CALL RHO2 (Y<integer 2>,Z<integer 1>)
            GOTO<label 2>
```

```
<label 1> Z<integer 3> = Z<integer 3>+ 1
          IF (Z<integer 3>.GT. ZLIM 25) CALL GVOVER (25, &<label 3>)
          YTEMP2 (Z <integer 3>) = Y <integer 2>
          CALL BDNO (Z <integer 4>,ZCPTR)
          IF (Z <integer 3> .LT. Z <integer 4>) GOTO <label 1 >
<label 2> Z <integer 5> = ZYY (ZYYPTR)
          Z <integer 6 > = Ø
<label 4 > Z <integer 6 > = Z <integer 6 >+ 1
```

An entry of the form

```
          YTEMP (Z <integer 6>+ Z <integer 5>)
```

is then stored on ITEMP.

5.6.3 The macro body for quad input

The quad input macro body is listed in Appendix 4.

To read in numerical data, the user must first specify the type of the data. The following types are possible:

(i) empty vector

(ii) scalar value

(iii) constant vector

The three types are distinguished by first specifying a value for the variable ZVBND. The values -1, Ø, n are used for cases (i) to (iii) respectively, where n is the number of elements in the constant vector.

The following code is generated:

```
           READ 1Ø2, ZVBND
           WRITE (6,1Ø2) ZVBND
           IF (ZVBND.NE.-1)  GOTO <label 1 >
           MARKER  =  -3
           WRITE (6,1Ø6)
           GOTO <label 2 >
<label 1 > IF (ZVBND.NE.Ø)  GOTO <label 3 >
           MARKER =  Ø
           READ (5,1Ø4) YTEM (1)
           WRITE (6,1Ø4) YTEM (1)
           GOTO < label 2>
<label 3 > IF (ZVBND.GT.ZLIM 26) CALL GVOVER (26, &<label 2>)
           READ (5,1Ø4)(YTEM (Zi),Zi = 1,ZVBND)
           WRITE (6,1Ø4)(YTEM (Zi), Zi = 1,ZVBND)
           ZCPTR = 1
           ZCBNDS (1)  =  ZVBND
<label 2 > Z <i+1 > =   Ø
           Z <i+1>  =  Z <i+1> + 1
```

An entry of the form Z<i+1> is stored on ITEMP to be re-accessed as required.

The FORMAT statements used are listed in macro number 16 in Appendix 4. They can be altered by the user if required.

Alternatively, the user can write his own input subroutines so that specification of ZVBND is unnecessary.

(Similar specifications of ZVBND are required for the quote-quad macro body.)

When quad or quote-quad input is used, the data supplied is printed out. This gives a closer approximation to the printed page produced when these instructions are used in APL .

CHAPTER VI

LABELS AND JUMPS

This chapter describes the handling of APL designational expressions. The method is much simplified by the presence of the lexical scanning phase. When the input APL code is lexically scanned, all label names are detected by the existence of a colon to their right. Entries (having type values of -4) are set up in NAMES. In addition, entries are set up in a two-dimensional table, LTABLE. A particular row of LTABLE holds the following information,

(i) The index for a label name in NAMES.

(ii) The corresponding row number for the statement in which the label occurs.

Entries are added to LTABLE sequentially.

Since the lexical scan is completed before code generation time, LTABLE will hold all the necessary information for production of GOTO statements. Forward jumps cause no problem because the appropriate label entries have previously been set up during the lexical scan.

All APL functions and subroutines have assumed label numbers. For example, consider the subroutine

```
    ∇F
[1] ----------
[2] ----------
      ⋮
      → 2
      ⋮
[n]
    ∇
```

The statement → 2 means "transfer control to the 2nd statement in subroutine F ".

To allow for this possibility, a count of the number of lines in a function or subroutine is maintained. In fact, there is a stack of line counts to handle sets of functions or subroutines. The line count is increased as required until the end of a function or subroutine is recognised. For each line processed, entries are set up in NAMES and LTABLE for the current line count value. The NAMES entries have type values of -4.

Thus, at code generation stage, the appropriate line numbers are produced at the start of the code corresponding to each line. For example, statement 2 of subroutine FN would be converted to

```
      2 CONTINUE
```

in the translated routine; similarly for all n lines in the subroutine. This allows for statements of the form

```
      → n
```

where n is not known until run-time of the converted routine.

Label numbers used in FORMAT statements and introduced during code generation have values starting from 100. Thus, up to 99 lines may be present in an APL function or subroutine. If more than 99 lines exist, then duplicate label numbers will be generated.

Now consider the statement

```
      → N
```

In general, N may be :

(i) a label name

(ii) a scalar variable name

(iii) a scalar constant with integral value

(iv) an empty vector

(v) a constant vector

(vi) a vector variable name

(vii) an expression with scalar result

(viii) an expression with empty vector result

(ix) an expression with non-scalar result

Each case is considered separately. The descriptions are given in §6.1 to §6.9 respectively. Cases (i) to (ix) are distinguished either by type value or by the value of MARKER.

6.1 Statements of the Form '→N', Where N is a Label Name

N has a type value of -4, and its NAMES index, I, is given by IDSTK (IDLPTR+1) .

The entries in the first column of LTABLE are searched sequentially until an entry is found having value I . If LTABLE (J,1) equals I, then the required label number is given by K = LTABLE (J,2) . Thus, it is only necessary to generate code of the form

```
GOTO K
```

6.2 Statements of the Form '→N', Where N is a Scalar Variable Name

N has a type value of Ø, and its NAMES index, I, is given by IDSTK (IDLPTR+1) . The value of N will not be known until run-time. Two possibilities may arise:

(1) $1 \leq N \leq MAX$, where MAX is the number of lines in the function or subroutine, or

(2) $N < 1$ or $N > MAX$.

For case (1) code of the form

```
GOTO <label number>
```

is required.

For case (2) the code

```
RETURN
```

should be generated.

The above two possibilities are catered for by generating code of the form

```
Z1  =  N
GOTO 1ØØØ
```

There is a corresponding switch

```
1ØØØ   IF (Z1.LE.Ø .OR. Z1.GT.MAX) RETURN
       GOTO (1,2,3,---,MAX), Z1
```

6.3 Statements of the Form '→N' Where N is a Scalar Constant With Integral Value

As in §6.2, N has a type value of Ø, and its NAMES index, I, is given by IDSTK(IDLPTR+1) . The value of N is known at code generation stage. However, distinguishing this case from that above would involve the introduction of a few more macro instructions. In fact, this case is treated exactly as that outlined above, and code of the form

```
Z1  =  N
GOTO 1ØØØ
```

is again generated.

6.4 Statements of the Form '→N', Where N is an Empty Vector

In this case there is a vacuous branch and sequential execution of statements is required. Thus no code is generated.

6.5 Statements of the Form '→N', Where N is a Constant Vector

N has a type value of -5 and its NAMES index, I, is given by IDSTK (IDLPTR+1) .

The required branch is determined by the first element of the constant vector. The first element value can be obtained (at run-time) by applying the function FUN to I with first parameter set to 1 . Thus, code of the following form is generated,

```
Z1  =  FUN (1,I)
GOTO 1000
```

The function FUN is present in module library SARUN .

6.6 Statements of the Form '→N', Where N is a Vector Variable Name

N has type value 1 and its NAMES index, I, is given by IDSTK (IDLPTR+1). The NAMES index for N can be used to provide the key and hence the dope vector address, J, for the vector N .

In this case the switch at label 1000 has to be indexed by the value of the first element of vector N . This value is given by YSTORE (KA) , where K = DOPES(J,3) , and KA = 10 * (K-1) + 1 . The required result is produced by generating code of the following form:

```
CALL VSET (I,K)
KA  =  10 * (K-1) + 1
Z1  =  YSTORE (KA)
GOTO 1000
```

Given the index I for the vector N , the subroutine VSET produces the result K, where K is the number of the first block of YSTORE associated with N . Subroutine VSET is listed in module library SARUN.

6.7 Statements of the Form '→N', Where N is an Expression With Scalar Result

In this case MARKER has value Ø and the scalar result code is stored in ITEMP . The following code is generated

```
Z1  =  < scalar result code >
GOTO 1ØØØ
```

6.8 Statements of the Form '→N', Where N is an Expression With Empty Vector Result

Here, MARKER has value -3 and a dummy result variable has been stored in ITEMP . The code

```
IF (MARKER.EQ.-3) GOTO < label>
```

is generated, where

```
< label> CONTINUE
```

is the last code line generated for a result.

6.9 Statements of the Form '→N', Where N is an Expression With Non-Scalar Result

This case has been limited to handle only expressions having a MARKER value of -5 . The index for the non-scalar result is set to 1 so that the first element of the result will be used to determine the branch. For cases described in §6.7 to §6.9, therefore, the following code is generated:

```
IF (MARKER.EQ.-3)  GOTO <label 1 >
IF (MARKER.NE.Ø) GOTO < label 2 >
Z1  =  <scalar result code>
GOTO 1ØØØ
```

```
<label 2>    Z_n  =  1
             Z1  =  <non-scalar result code>
             GOTO 1ØØØ
<label 1>    CONTINUE
```

Here Z_n is the index variable for the non-scalar result.

CHAPTER VII

PROCESSING OF INITIAL INFORMATION

This chapter describes the processing of the initial information supplied with a set of APL routines. From the initial information, entries of the following form were stored for numeric or literal non-scalars. Three possibilities exist:

(a) if no further information was supplied for the non-scalar, the following entries are stored:

1. the position, I, of the non-scalar in the initial list,
2. the address, A, of the dope vector entry for the non-scalar,
3. $\emptyset$;

(b) if partial information was supplied for the non-scalar, the following entries are stored:

1. the position, I, of the non-scalar in the initial list,
2. the address, A, of the dope vector entry for the non-scalar,
3. -N, where N is the number of dimensions of the non-scalar;

(c) if full information was supplied for the non-scalar, the following entries are stored:

1. the position, I, of the non-scalar in the initial list,
2. the address, A, of the dope vector entry for the non-scalar,
3. N, where N is the number of dimensions of the non-scalar,
4. b_1, where b_1 is the bound for the first dimension of the non-scalar

⋮

4+N-1. b_N, where b_N is the bound for the N^{th} dimension of the non-scalar.

All entries stored are in I6 format.

Initially, one block of space in YSTORE was set aside for storage of each numeric non-scalar for which no additional information was provided.

The generated code may contain calls of a subroutine which allocates or de-allocates storage for a particular non-scalar. This will be the case if the dyadic 'rho ' function appears in the original routine. However, before running the converted routines, the initial information is taken into account. Where possible, contiguous blocks of YSTORE are allocated for non-scalars. This is discussed later.

For non-scalars with elements stored in contiguous blocks, a fixed amount of storage has been set aside in YSTORE. This amount is the maximum amount of space required by the non-scalar. Thus, for non-scalars with elements stored in contiguous blocks, the amount of space allocated should not be varied dynamically. The dimensions for the non-scalar may, however, vary dynamically. It is therefore arranged that the storage allocation subroutine has no effect for non-scalars stored in contiguous blocks of YSTORE. The current dimensions are updated as required throughout the running of the converted routine.

The dope vector table is now updated (using the initial information) so that those non-scalars having elements stored in contiguous blocks can be detected by examining the table. The table is updated in the following manner:

(a) For arrays with full information given, the actual start address, SA, in YSTORE can be calculated. DOPES (n,3) and DOPES (n,4) are set to Ø and SA respectively, where n is the address of the dope vector entry.

(b) For arrays with no information given, the dope vector entry is unaltered.

(c) For arrays with only partial information given, the negative of the position of the array in the initial list is stored in column 3.

For cases (a) and (c) above, the fifth column of the dope vector entry is filled in at this stage. For (a) the bound information is stored in array ZBONDS, from position ZBPTR onwards. ZBPTR is then increased as required. For (c) storage is set aside in ZBONDS for the bound information and ZBPTR is set up as required. In this case the bound information cannot be filled in until it is obtained at run-time of the converted routine.

Now consider arrays with full or partial additional information supplied. To access the I^{th} element, relative to the base address, the following steps are sufficient.

1. Obtain the start address, SA, for the non-scalar from the dope vector table.
2. Obtain the actual address, AA, which is given by SA + I .

Thus the time-consuming accessing method used in the function FIND can be replaced by a much simpler function. A method of optimising the generated code by removing unnecessary FIND calls is outlined in Chapter VIII.

The following types of entry are thus present in DOPES at this stage:

(a) <u>Column 3 > Ø</u>

No additional information has been given. The accessing method used in the function FIND is essential.

(b) <u>Column 3 < Ø</u>

Partial information has been supplied for the non-scalar, and the DOPES entry is in an intermediate form. It will be converted to form (c) using information supplied by the user at run-time of the converted routine. This is discussed in §7.1 . In this case the generated code contains an unnecessary call of the function FIND.

(c) <u>Column 3 = Ø and column 4 ≠ Ø</u>

Full information has been supplied and the dope vector entry contains the exact start address for the non-scalar (in column 4). Again the code generated contains an unnecessary call of the function FIND.

For example, suppose that the following information was supplied initially:

A	Ø	3	4	4	3
B	Ø	Ø			
C	Ø	-1			
D	Ø	2	1Ø	2	
E	Ø	-2			

Thus, the total number of blocks of YSTORE required for A is 5, for D is 2, and it is known that C is 1-dimensional and E 2-dimensional. No further information is given for B .

During processing, the dope vector entries for A, B, C, D and E are altered to the form shown in Diagram 7(a). The corresponding layout of YSTORE is indicated in Diagram 7(b). It is assumed that ZBPTR has value 1Ø initially. Dotted lines have been used to indicate that the exact amount of space for non-scalars C and E is not known at this stage.

The amount of space allocated for B may be updated dynamically if B has been re-dimensioned in the APL routine. The amount of space allocated for A, C, D and E, however, will remain constant during execution of the converted routine.

At this stage, start addresses for A and D are known exactly, and

1		Ø	n	3	1Ø
2		2	2		
3		-3		1	13
4		Ø	m	2	14
5		-5		2	16

Diagram 7(a) : Shows DOPES entries for non-scalars A,B,C,D,E after the initial information has been processed.

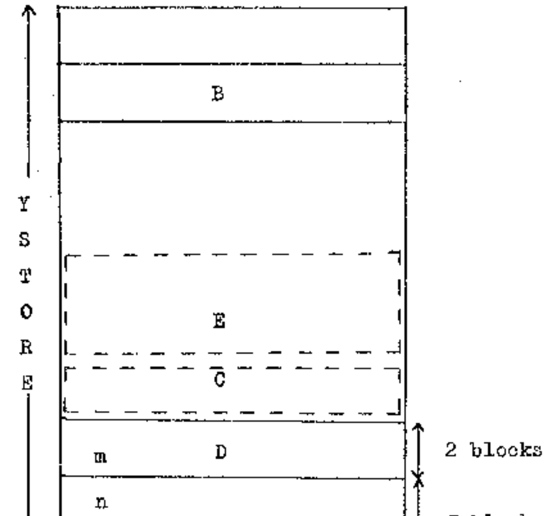


Diagram 7(b) : Shows layout of YSTORE corresponding to DOPES entries as given in Diagram 7(a).

the start addresses for C and E can be calculated immediately before the converted APL routine is run.

The generated code can now be optimised. The optimisation process (with particular reference to the removal of unnecessary FIND calls) is described in Chapter VIII. In replacing FIND calls it is assumed that the exact start address is stored in column four of the DOPES entry for any non-scalar. This will be the case even for partially specified non-scalars, as the DOPES entries are altered to this form immediately before execution of the converted routines.

The processing of the information supplied at run-time of the converted routines is discussed in §7.1 .

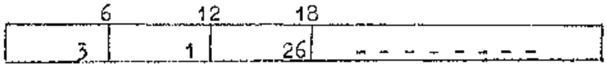
7.1 Handling of Bounds Information for Partially Specified Numeric Non-scalars

Corresponding to each partially specified numeric non-scalar in the initial list, the following information must be supplied by the user at run-time:

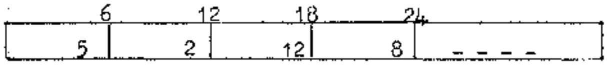
1. The position, I, of the non-scalar in the initial list.
2. The number, N, of dimensions of the non-scalar.
3. N entries containing the bounds for each dimension.

For example, referring to Diagram 7(b), if YSTORE has 5000 locations, then n = 4951 and m = 4931 .

Suppose the information specified at run-time for non-scalars C and E is as shown below:

(i) 

for C

(ii) 

for E

Then the start address of C can be calculated as:

$$4931 - (26 + 9) \div 10 \times 10 = 4901$$

Similarly, the start address for E can be calculated as:

$$4901 - ((12 \times 8) + 9) \div 10 \times 10 = 4801 .$$

That is, in general

$$SA_{n+1} = SA_n - (\text{no. of elements} + 9) \div 10 \times 10 ,$$

where SA_1 is the lowest start address for the fully specified non-scalars.

For each line of information obtained, the following process is carried out:

The array DOPES is scanned, in particular column 3. Suppose (for entry i in the dope vector table)

(a) Column $3 > 0$

This corresponds to a non-scalar for which no additional information has been provided. No action is required. Entry (i + 1) in DOPES is now scanned.

(b) Column $3 = 0$

Examine column 4. If column $4 = 0$, then the entry is either empty or corresponds to a literal non-scalar. Again no action is required and entry (i + 1) is now tested. If column $4 \neq 0$, then the entry is for an array with full information provided initially. The dope vector entry has already been filled correctly, and the bounds have been set up in ZBONDS. No further action is required and entry (i + 1) is now tested.

(c) Column $3 < 0$

Suppose column $3 = -k$. Then the bounds information for the k^{th} non-scalar in the initial list must now be accessed. Using this information the exact start address for the non-scalar can be calculated. Then DOPES (i,3) is set to Ø and DOPES (i,4) to the start address.

Suppose DOPES (i,5) = N and
DOPES (i,6) = M ,

then the N bound values are stored in ZBONDS, starting from position M . This is illustrated in Diagram 7.1(a) .

It is assumed at this stage that information is read in in increasing order of position in the initial list. Thus, for example, the information for C appeared before the information for E .

The run-time bound information is supplied in I6 format. Therefore the first 6 columns of each card could be scanned and the data reordered accordingly.

When the initial information was first obtained, the address, I, of the dope vector entry for each non-scalar was stored. This value has not been used in updating the entries in the dope vector table. It is used, however, in replacing unnecessary FIND calls. The value DOPES (I,2) has to be compared with the fourth parameter of each FIND call recognised. If a match is found, then column 3 has to be tested for a value $\leq$ Ø, indicating that the FIND call can be replaced. This is discussed in greater detail in Chapter VIII.

In conclusion, the entire conversion process is summarised below.

1. Partially process initial information.
2. Convert APL routines to target language code. This involves:

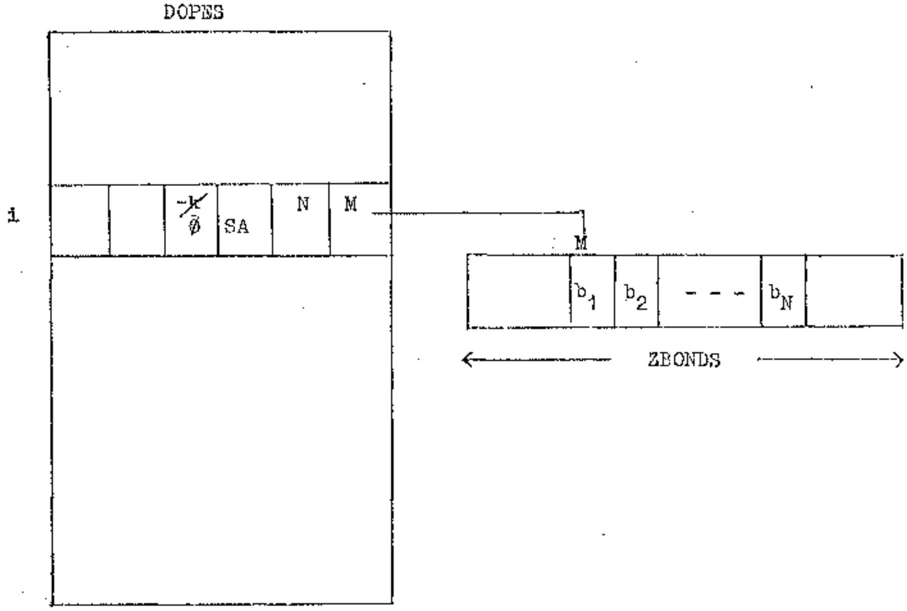


Diagram 7.1(a) : Shows setting up of entries in ZBONDS using columns 5 and 6 of the DOPE vector table.

2.1 lexical scan,

2.2 right-to-left scan and production of intermediate code,

2.3 left-to-right scan and production of parameter information on a stack,

2.4 macro expansions.

3. Use initial information to update the dope vector entries as described.

4. Optimise code, in particular by replacing unnecessary FIND calls by simpler functions.

5. Read in bounds for partially specified arrays and update the dope vector entries.

6. Run the converted routine.

CHAPTER VIII

OPTIMISATION OF CODE

This chapter describes optimisation of the generated code. The main topic is the replacement of unnecessary "FIND" calls by simpler functions. This is discussed in §8.1 and §8.2 . In §8.3 other means of optimising the generated code are described.

It should be emphasised that no attempt was made at code generation time to do any optimisation of code. This was reserved for a scan of the code after generation. For this reason, the generated code is in parts very inefficient, but can be altered to give a considerable increase in efficiency.

Another factor contributing to the initial inefficiency of the generated code is the generality demanded by APL . It is this factor which necessitates the production of a very large amount of code corresponding to only a simple APL expression. One striking example is provided by the APL expression +/A, which is discussed in more detail in §8.1.9 .

In producing the generated code, two courses were available for the handling of global array declarations. These were:

1. to use the exact values for all array bound specifications and to place the arrays in the COMMON area for each subroutine or function;

2. to give all arrays unit dimensions and to place the array names in the parameter list of every subroutine and function containing references to the arrays.

The first method is more efficient as parameter linkage, which can be a costly operation, requires less time and storage space in this case. However, the second method has the advantage that less alterations need be made to the code to change the bound values for some arrays, should this be found necessary. (Only the main program and the subroutine ZINIT, in which global non-scalars are initiallised, need be altered using the second method.) With this aim in view, the second method was used in generating the code, but the first method is employed in all the examples given, as it provides more readable code. In addition, it is easier to generate ALGOL or PL/1 instead of FORTRAN if COMMON statements are not used, (see Appendix 7).

8.1 Replacement of Unnecessary "FIND" Calls

APL allows great diversity in accessing of non-scalars. Some examples are given below, illustrating a number of different types of non-scalar access. The code generated is not always very efficient, as this inherent diversity must be allowed for.

Calls of the function FIND are contained in subroutines FIND1 and SPECS. These subroutines, therefore, must also be replaced by simpler functions, where possible. At the same time, all the essential information must be retained (in setting up current bounds, for example). For this reason, calls of the subroutines

1. ZADDR
2. ELPERM

are often generated when a "FIND" call is being replaced. (The notation "FIND" is used to mean FIND or FIND1 or SPECS .) These subroutines have the following functions:

1. ZADDR (I, ZST, ZNUM, ZBOUND)

calculates (a) the start address ZST

(b) the number of elements ZNUM

(c) the pointer, ZBOUND, to the bound information

for the non-scalar with NAMES index I .

2. ELPERM (I, J, K, L, N, Z)

reorders the subscript values in array Z according to the value of I . For a FIND or FIND1 call, the values I, J and K correspond to the first 3 parameters. For a SPECS call, I, J and K all have value Ø . Z is a two-dimensional array, each row giving the subscript values for a particular level in nested subscript expressions. L defines the row of Z to be accessed, and N the number of elements within the Lth row. The current bounds for a non-scalar expression are also updated in ELPERM, in the manner specified by the value of I .

The following types of array access are now considered.

1. Accessing an element of a 1-dimensional array
2. Accessing an element of a 2-dimensional array
3. Accessing all the elements of an n-dimensional array
4. Addition of an element of a 1-dimensional array to all the elements of an n-dimensional array
5. A nested subscript expression
6. An expression involving non-scalar subscripts
7. Accessing all the elements of a particular column of an array
8. An expression involving a constant vector subscript
9. An expression involving the APL reduction operator.

EXAMPLE 8.1.1

Consider the APL expression

$A[I + 1]$

where I is scalar and A is non-scalar.

The code generated corresponding to the above expression is of the form shown below.

```
Z∅   =   ∅
ZB1  =   ZPOINT (ZPT)
ZPT  =   ZPT + 1
ZINDX (ZB1 + 1)  =  I + 1
ZPOINT (ZPT)   =   ZB1 + 1
CALL STARTS (A_NAMES, Z1, Z2, ZNC )
CALL FIND1 (FP1, FP2, FP3, A_NAMES, Z3, Y3, Z2, ZNC )
ZPT   =   ZPT - 1
```

The code Y3 is then stored on ITEMP to be used as parameter for any further macro expansions.

The above code merits further discussion. It may be observed that the variable Z∅ is set to ∅ initially and is not referred to thereafter. It was included to handle cases where the subscript expression is non-scalar. Thus it is redundant for all scalar subscript expressions, but not for non-scalar subscript expressions,in which it is used to handle the implicit looping operations (see EXAMPLES 8.1.6, 8.1.7 and 8.1.8) .

As was discussed in Chapter I, §1.2.5, the subscript expressions are stored in the array ZINDX . Since nested subscripts are allowed (see EXAMPLE 8.1.5), it is essential at any time only to access the current level of ZINDX to obtain the subscript expressions. For this reason, the stack, ZPOINT, was introduced, and the top-most positions of ZPOINT define the section of ZINDX to be accessed at any time.

The subroutine STARTS simply does some preliminary calculations before FIND1 is executed. The first parameter, A_{NAMES}, is the NAMES index for A . (A similar notation will be used throughout.) FIND1 calculates the index, Z3, in YSTORE and the value, Y3, for the non-scalar being accessed. The first three parameters determine the type of accessing required and the type value and index (or value, if the type value is Ø) for the last operand for dyadic operations. In addition to the normal type of array accessing, there are 14 other types, corresponding to certain APL mixed functions. These are listed in Chapter I, §1.2.5 .

EXAMPLE 8.1.2

Consider a similar example

$$A \longleftarrow B\left[I \;;\; J+6\right] + 1$$

where B is non-scalar and A, I, J are scalar. Then code of the following form is generated,

```
Z1    =   Ø
ZB1   =   ZPOINT (ZPT)
ZPT   =   ZPT + 1
ZINDX (ZB1 + 1)   =   I
ZINDX (ZB1 + 2)   =   J + 6
ZPOINT (ZPT)   =   ZB1 + 2
CALL STARTS (B_NAMES ---- )
CALL FIND1 (--- B_NAMES --- Y3 --- )
ZPT   =   ZPT - 1
A     =   Y3 + 1
```

Now consider optimisation of the previous examples. Assume, firstly, that full information is available for A in EXAMPLE 8.1.1 . Then

$$\text{DOPES } (i_A , 3) = \emptyset$$

and $$\text{DOPES } (i_A , 4) = S_A$$

where i_A is the address of the dope vector entry for A and S_A is the start address in YSTORE for A . The elements of A are stored in consecutive locations of YSTORE, starting from position S_A .

The code for A [I+1] could thus be replaced by code of the form

$$Y3 = \text{YSTORE (DOPES } (i_A,4) + (I+1) - 1)$$

and the form Y3 is still suitable as the next operand.

Now assume that B has full information specified in EXAMPLE 8.1.2 . Then, if Y3 is set to

$$\text{YSTORE } (S_B + [(J+6) - 1] * b_1 + I - 1)$$

where B is a $(b_1 * b_2)$ array and S_B is the start address in YSTORE for B, the code Y3 is still equivalent to B [I ; J+6] .

The expression for the equivalent YSTORE element is obviously dependent on the number of dimensions of the original non-scalar. The above code can be generalised as shown below to handle n-dimensional array accesses.

```
(A)        Z1    =   Ø
           Z(i,1)   =  < 1st parameter expression >
             ⋮
           Z(i,n)   =  < nth parameter expression >
           CALL ELPERM (- - -)
           Z2    =   n - 1
           ZPROD   =   Z(i,n)
```

```
        CALL ZADDR (I,ZST,ZNUM,ZBOUND)
1ØØ1    IF (Z2.LE.Ø) GOTO 1ØØ2
        ZPROD  =  (ZPROD-1) * ZBONDS (ZBOUND + Z2-1) + Z(i,Z2)
        Z2  =  Z2-1
        GOTO 1ØØ1
1ØØ2    Y3  =  YSTORE (ZST + ZPROD-1)
```

Here I is the NAMES index for the non-scalar being accessed. It is assumed that the ith nested level is being dealt with.

If n is 1, then code of the following form is sufficient.

```
(B)     Z1  =  Ø
        Z(i,1)  =  < 1st parameter expression >
        CALL ELPERM (- - -)
        ZPROD  =  Z(i,1)
        CALL ZADDR (I,ZST,ZNUM,ZBOUND)
1ØØ2    Y3  =  YSTORE (ZST + ZPROD-1)
```

Here ZPROD and the label 1ØØ2 could have been eliminated. They are used merely to provide conformity with case (A).

In (A), variables Z2 and Y3 have been used, replacing the use of these names in the original code. It is essential to avoid ambiguity when introducing variable names. Use of the name ZPROD causes no ambiguity, as ZPROD would have been replaced by YZPROD, if it had appeared in the APL text.

The label numbers used must also be distinct.

In examples such as those given above, the replaceable code is delimited by

ZB < integer > = ZPOINT (ZPT) ... (a)

and ZPT = ZPT - 1 ... (b)

Thus, after recognising a statement of the form (a), the subsequent code must be scanned for a replaceable "FIND" call.

In addition, if a statement of the form

ZINDX (---) = < expression >

is recognised, then the code for < expression > must be retained. The number, n, of such expressions must also be noted.

To test for a replaceable "FIND" call the following action is required:

1. Scan column 2 of the dope vector table DOPES for an entry equal to the fourth parameter of the FIND call.

2. If there is no equal entry, the "FIND" call is necessary and the scan of of the generated code should be resumed.

3. If there is an equal entry, the "FIND" call is unnecessary and may be replaced.

There may be nesting of non-scalar references in APL, and correspondingly nesting of the types of statement delimited by forms (a) and (b) above. It is thus necessary to maintain a count, c, of the "current level of complexity". The value of c should be incremented by 1 whenever a statement of form (a) is recognised, and decremented whenever a statement of form (b) is recognised.

The method of transforming the generated code to its reduced form is outlined in §8.2 .

EXAMPLE 8.1.3

Consider

$$X \longleftarrow Y + Z$$

where **X**, Y and Z are $(b_1 * b_2 * --- * b_n)$ arrays.

To handle this expression it is necessary to set up loops, as every element of the non-scalars is accessed in turn. The code required for the looping operations is fairly complex, as it must allow for n-dimensional arrays, where n is not known at the code generation stage. The generated code is of the following form,

```
           ZB1   =   ZPOINT (ZPT)
           ZPT   =   ZPT + 1
           CALL STARTS (Y_{NAMES},Z1,Z2,--- )
           ZPOINT (ZPT)   =   ZB1 + Z2
           Z3  =  1
1ØØ        Z4  =  ZB1 + Z3
           ZINDX (Z4)  =  1
           Z3  =  Z3 + 1
           IF (Z3.LE.Z2) GOTO 1ØØ
           Z5  =  ZB1 + Z2
           Z6  =  Z2 - 1
           ZSAVE  =  Ø
1Ø1        CALL FIND1 (--- Y_{NAMES} --- Y_{n} --- )
           CALL FIND1 (--- Z_{NAMES} --- Y_{n+1} --- )
           Y7  =  Y_{n} + Y_{n+1}
```

```
          CALL SPECS (X_NAMES,Y7,Z7)
          ZSAVE  =  1
          ZINDX (Z5)  =  ZINDX (Z5) + 1
          IF (ZINDX (Z5).LE.ZCBNDS (ZCPTR) )  GOTO 1Ø1
1Ø2       Z8  =  Z6 + 1
1Ø3       ZINDX (Z8 + ZB1)   =   1
          Z8  =  Z8 + 1
          IF (Z8.LE.ZCPTR) GOTO 1Ø3
          IF ((ZB1 + Z6).LE.Ø)  GOTO 1Ø6
1Ø4       ZINDX (ZB1 + Z6)  =  ZINDX (ZB1 + Z6) + 1
          IF (ZINDX (ZB1 + Z6).LE.ZCBNDS (Z6))  GOTO 1Ø5
          IF (Z6.EQ.1)  GOTO 1Ø6
          ZINDX (ZB1 +Z6)   =   1
          Z6  =  Z6 - 1
          GOTO 1Ø4
1Ø5       Z6   =   Z2 - 1
          GOTO 1Ø1
1Ø6       ZPT   =   ZPT - 1
          CALL SPECB
```

The subroutine SPECS handles specifications for all possible type combinations other than the simple case

< scalar> = < scalar >

However, for the case < vector > = <expression>, the YSTORE elements containing < vector> are not updated immediately. If they were, problems could arise with statements such as:

Y ⟵ 1,Y

In this case, the first element (1) of the right-hand side would be stored in Y(1). Then the next element (Y(1)) of the right-hand side would be stored in Y(2). It can be seen that Y(1) should not have been updated before its value was stored in Y(2).

This problem is counteracted in the following way. Each time SPECS is called in the loops, the index of the YSTORE element to be updated, together with the new value, is stored. Then, on completion of the loops, SPECB is called, and the appropriate YSTORE elements are then updated.

The variable ZSAVE is Ø the first time round the loops, and is 1 at all other times. Its value is tested in FIND to determine whether the current bounds are to be updated.

Here there is a group of FIND calls in one level of complexity. The relevant lines of code should only be removed once, if at all. If there is any one "FIND" call which is not to be replaced, then no lines of code should be deleted. The relevant "FIND" calls should still be replaced, however.

It is possible to reduce the code very considerably if all the "FIND" calls in one level of complexity are replaceable. For example, consider the following case.

(a) <u>X, Y and Z all replaceable</u>

The generated code may be replaced by:

ZSAVE = Ø

CALL ZADDR (X_{NAMES},ZST ---)

Z_{n+2} = ZST - 1

Z3 = 1

CALL ZADDR (Y_{NAMES},ZST,---)

Z_m = ZST - 1

CALL ZADDR (Z_{NAMES},ZST,ZNUM,---)

```
        $Z_{m+1}$ = ZST - 1
1ØØ     $Z_m$ = $Z_m$ + 1
        $Y_n$ = LPERM ($Z_m$,I,J,K,L)
        $Z_{m+1}$ = $Z_{m+1}$ + 1
        $Y_{n+1}$ = LPERM ($Z_{m+1}$,I,J,K,L)
        $Y_7$ = $Y_n$ + $Y_{n+1}$
        $Z_{n+2}$ = $Z_{n+2}$ + 1
        CALL SPECA ($Z_{n+2}$,$Y_7$)
        ZSAVE = 1
        Z3 = Z3+1
        IF (Z3.LE.ZNUM) GOTO 1ØØ
        CALL SPECB
```

In SPECA, the YSTORE element to be altered (together with the new value) is stored and the changes made later by calling SPECB.

The function LPERM uses the input parameters I, J, K and L (the first 4 parameters of a FIND call) to calculate the index N in YSTORE for the required element. The value YSTORE (N) is returned. In addition, if ZSAVE is Ø, then the current bounds (ZCBNDS) will be updated if necessary.

LPERM need not be called if a call of SPECS is being replaced, as normal accessing is then required in the FIND call. For the vast majority of cases, there will be normal accessing of non-scalars, and thus LPERM will have no effect.

There is still an increase in efficiency if only some of the "FIND" calls in a group are replaceable. This is not so readily apparent, however, as all the looping statements are still required.

A method of transforming the generated code in the above manner is discussed in § 8.2 .

EXAMPLE 8.1.4

Consider

$$A \longleftarrow B + C\,[6]$$

where A and B are n-dimensional arrays. In the generated code, the code corresponding to C [6] is produced first, followed by an array access of the type given in EXAMPLE 8.1.3 . Thus, where appropriate, the generated "FIND" calls can be replaced by code of the form given in EXAMPLES 8.1.1 and 8.1.3 .

An example of the nesting of subscript expressions is now given.

EXAMPLE 8.1.5

Consider the APL statement

$$E \longleftarrow A\,[B\ ;\ C\,[D]\,]$$

where A and C are non-scalars and E, B and D are scalars. The code generated for the above statement will be of the form shown below.

```
Z1   =   ∅
ZB1  =  ZPOINT (ZPT)
ZPT  =  ZPT + 1
ZINDX (ZB1 + 1)   =   B
ZB2  =  ZPOINT (ZPT)           code for the non-scalar access  C[D]
 ⋮                             (see EXAMPLE 8.1.1)
ZPT  =  ZPT - 1                Y_n ≡ C [D]
ZINDX (ZB1 + 2)  =  Y_n
ZPOINT (ZPT)     =  ZB1 + 2
CALL STARTS (A_NAMES --- )
CALL FIND1 (--- A_NAMES --- Y_n+1 --- )
ZPT  =  ZPT -1
E    =  Y_n+1
```

If C is replaceable, then the corresponding code for C [D] may be reduced as described for EXAMPLE 8.1.1 .

If A is also replaceable, the following reduction is possible,

```
Z1      =   Ø
Z(1,1)  =   B
```

< reduction of type given for EXAMPLE 8.1.1 corresponding to C [D] >

```
Z(1,2)  =   Y_n
  .
  .
  .
```

< remainder of reduction as for EXAMPLE 8.1.2 corresponding to A [-----] >

```
E   =   Y_n+1
```

An example illustrating the handling of vector subscripts is now given. Non-scalar subscripting is catered for, but the subscript expression must be only one-dimensional. The method employed does not exclude n-dimensional subscripting (n > 1), but higher dimension subscripting has been excluded simply to make the generated code less unwieldly.

EXAMPLE 8.1.6

Corresponding to the APL statement

```
E ←— A [B + C [D] ]
```

where E, A, B and C are non-scalar, code of the following form is generated:

```
        Z1    =   Ø
        ZB1   =   ZPOINT (ZPT)
        ZPT   =  ZPT + 1

   <code similar in form to that generated for EXAMPLE 8.1.1 >

        ZB2     =   ZPOINT (ZPT)   --|
                .                    |       as for EXAMPLE 8.1.3
                .                    |
                .                    |
        ZSAVE  =   Ø               --|
1Ø1     CALL FIND1 (--- $B_{NAMES}$ --- $Y_{n+1}$ --- )
        Z8    =   ZY (ZYPTR)
        Z1    =   Z1 + 1
        ZTEMP (Z8 + Z1)   =   $Y_{n+1}$ + $Y_n$
        ZSAVE   =   1              --|
          .                          |
          .                          |       as for EXAMPLE 8.1.3
          .                          |
1Ø6     ZPT     =  ZPT - 1         --|
        ZYPTR  =  ZYPTR + 1
        ZY (ZYPTR)  =   Z1
        Z1Ø     =  ZY (ZYPTR - 1)
        Z11     =  Ø
1Ø7     Z11     =  Z11 + 1
        ZINDX (ZB1 + 1)   =   ZTEMP (Z1Ø + Z11)
        ZPOINT (ZPT)      =   ZB1 + 1
        CALL STARTS ( $A_{NAMES}$ - - - )
        CALL FIND1 (--- $A_{NAMES}$ --- $Y_{n+2}$ --- )
        Y12   =   $Y_{n+2}$
        CALL SPECS ( $E_{NAMES}$, Y12, --- )
        CALL BDNO (Z13,ZCPTR)
        IF (Z13.LT.Z11)  GOTO 1Ø7
        CALL SPECB
        ZPT   =   ZPT - 1
```

In this example, the variable Z1 is not redundant. The subscript values are stored in the next level of ZTEMP, Z1 being incremented each time round the loops so that successive elements of the non-scalar subscript will be stored in successive elements of ZTEMP. The appropriate elements of ZTEMP are then stored in turn in the correct location of Z1NDX, and code generation continues in the usual way.

A similar technique is used in all cases of vector subscripting.

For the above expression, the generated code can be optimised in a number of different ways, depending on the amount of information supplied for the non-scalars. The sections of code of the forms given for EXAMPLES 8.1.1, 8.1.2 and 8.1.3 can be optimised, where appropriate, to the forms described previously. Code which cannot be placed in one of the above sections is <u>always</u> non-replaceable, and also appears in the optimised version.

The possible transformations of the above code may also be effected using the method given in §8.2 .

EXAMPLE 8.1.7

Consider the APL statement

```
        B ⟵ A [; C ]
```

where B and A are non-scalar, and C is scalar.

In this case, as in all cases of vector subscripting, the generated code is unwieldy. This is unavoidable, since the implied looping operations must be taken into account.

Code of the following form is generated corresponding to the above statement.

```
          Z1    =   Ø
          ZB1   =  ZPOINT (ZPT)
          ZPT   =  ZPT + 1
          Z2    =  ZY (ZYPTR)
          CALL BDVAL (A_NAMES,ZDIMNO,Z3)
1ØØ       Z1    =   Z1 + 1
          ZTEMP (Z1 + Z2)   =   Z1
          IF (Z1.LT.Z3)  GOTO 1ØØ
          ZYPTR    =    ZYPTR + 1
          ZY (ZYPTR)   =   Z1
          Z4    =    ZY (ZYPTR - 1)
          ZSAVE    =   Ø
          MARKER   =   -5
1Ø1       Z5    =    Ø
1Ø2       Z5    =    Z5 + 1
          ZINDX (ZB1 + 1)    =    ZTEMP (Z4 + Z5)
          ZINDX (ZB1 + 2)    =    C
          ZPOINT (ZPT)    =    ZB1 + 2
          CALL STARTS (A_NAMES - - - )
          CALL FIND1 (--- A_NAMES --- Y_n --- )
          ZSAVE    =   1
          Y6    =    Y_n
          CALL SPECS (B_NAMES,Y6,Z5)
          CALL BDNO (Z7,ZCPTR)
          IF (Z5.LT.Z7)  GOTO 1Ø2
          CALL SPECB
          ZPT    =    ZPT - 1
```

The subroutine BDVAL (I,J,K) has input parameters I and J, and output parameter K, where

1. I is the index of the array in NAMES
2. J is the dimension number
3. K is the bound value for dimension number J .

Again, any statements of a form not found in EXAMPLES 8.1.1, 8.1.2 and 8.1.3 are considered to be irreplaceable. All other sections of code are reduced, where possible, to the forms given for the above examples.

Thus, if both A and B are replaceable, the generated code can be reduced to the form shown below.

```
          Z1       =   Ø
          CALL ZADDR (B_NAMES,ZST,ZNUM,ZBOUND)
          Z_n+1    =   ZST - 1
          Z2       =   ZY (ZYPTR)
              :
              :
              :
1Ø2       Z5       =  Z5 + 1
```

< reduction of the form given for EXAMPLE 8.1.2 >

```
          ZSAVE    =   1
          Y6       =   Y_n
          Z_n+1    =   Z_n+1 + 1
          CALL SPECA (Z_n+1,Y6)
          CALL BDNO (Z7,ZCPTR)
          IF (Z5.LT.Z7)  GOTO 1Ø2
          CALL SPECB
```

At first sight, it may appear that the reduced code is even lengthier than the original. However, all the functions and subroutines referenced are much simpler in the reduced version.

§ 8.2 describes the method of carrying out the possible reductions for the above example.

EXAMPLE 8.1.8

Consider the APL statement

B ⟵ A[3 4 5]

There are implicit looping operations in the above statement. Again the subscript values are stored in successive positions in the array ZTEMP, and loops are set up to store the appropriate ZTEMP elements, in turn, in the correct subscript positions in ZINDX. The result is a vector whose three elements are stored in vector B . The code generated for the above statement is again unwieldy on account of the implied looping operations. However, considerable reductions are possible if additional information is available for A or B or both. The generated code is of the form shown below.

```
        Z1    =  Ø
        ZB1   =  ZPOINT (ZPT)
        ZPT   =  ZPT + 1
        ZB2   =  ZPOINT (ZPT)             ┐
          .                               │
          .                               │  as for EXAMPLE 8.1.3
          .                               │
        ZSAVE  =  Ø                       ┘
1Ø1     CALL FIND3 (--- A_NAMES --- Y_n --- )
        Z7    =  ZY (ZYPTR)
        Z1    =  Z1 + 1
        ZTEMP (Z7 + Z1)  =  Y_n
        ZSAVE  =  1                       ┐
          .                               │
          .                               │  as for EXAMPLE 8.1.3
          .                               │
1Ø6     ZPT    =  ZPT - 1                 ┘
        ZYPTR  =  ZYPTR + 1
        ZY (ZYPTR)  =  Z1
        ZSAVE  =  Ø
        MARKER  =  -5
```

```
1Ø7      Z9  =  Ø
1Ø8      Z9  =  Z9 + 1
         ZINDX (ZB1 + 1)  =  ZTEMP (Z7 + Z9)
         ZPOINT (ZPT)  =  ZB1 + 1
         CALL STARTS ($A_{NAMES}$ - - - )
         CALL FIND1 (--- $A_{NAMES}$ --- $Y_{n+1}$ --- )
         ZSAVE  =  1
         Y1Ø   =  $Y_{n+1}$
         CALL SPECS ($B_{NAMES}$,Y1Ø,Z9)
         CALL BDNO (Z11,ZCPTR)
         IF (Z9.LT.Z11)  GOTO 1Ø8
         CALL SPECB
         ZPT  =  ZPT - 1
```

The subroutine FIND3 is similar in concept to FIND1, the difference being that a constant vector element (obtained from NAMES) is produced as a result instead of an element of YSTORE. Constant vectors (and literals) may also be accessed in a number of different ways, corresponding to 14 of the APL mixed functions.

Some optimisation of the forms illustrated previously may be possible.

EXAMPLE 8.1.9

Consider the APL expression

+/A

where A is a numeric non-scalar. Even for such a simple expression, a large amount of code must be generated to allow for all the possible types of A . If A is a vector, then a scalar result is obtained; if A is an n-dimensional array, then an (n-1) dimensional result is obtained.

The number of dimensions of A is not known at code generation time. It is thus necessary to set up loops to handle n-dimensional array accessing, where n is unknown.

For the above reasons, the code generated for +/A is extremely unwieldy. There is no way round this since the additional information supplied for non-scalars is not taken into account at code generation time. However, once the additional information is considered, it is immediately possible to make a drastic reduction in the generated code. This is discussed below.

The generated code is first described. It takes the form shown below.

```
        CALL BDINFO (A_NAMES,ZDIMS,ZBOUND)
        IF (ZDIMS.NE.1)  GOTO 1Ø1
        ZB1   =   ZPOINT (ZPT)
        ZPT   =   ZPT + 1
        ZINDX (ZB1 + 1)   =   1
        OPL   =  < identity element for  + >
        ZPOINT (ZPT)   =   ZB1 + 1
1ØØ     Z2    =   FIND (1,Ø,Ø,A_NAMES - - - )
        OPR   =   YSTORE (Z2)
        OPL   =   OPR + OPL
        ZINDX (ZB1 + 1)   =   ZINDX (ZB1 + 1) + 1
        IF (ZINDX (ZB1 + 1).LE.ZBONDS (ZBOUND))  GOTO 1ØØ
        ZPT   =   ZPT - 1
        MARKER   =   Ø
        Z3    =   ZYY (ZYYPTR)
        Z4    =   1
        YTEMP (Z3 + Z4)   =   OPL
        GOTO 1Ø2
```

```
1Ø1     ZB1    =   ZPOINT (ZPT)
           .
           .                              as for EXAMPLE 8.1.3
           .
        ZSAVE  =  Ø
        CALL BNDSET (1, $A_{NAMES}$, 1)
        Z1Ø    =   Ø
        IF (ZCOORD (ZCDPTR).EQ.Ø) ZCOORD (ZCDPTR)   =   ZDIMS
        ZCD    =   ZCOORD (ZCDPTR)
        Z11    =   ZYY (ZYYPTR)
        OPL    =  < identity element for  +>
1Ø5     Z12    =   FIND (1,Ø,Ø, $A_{NAMES}$ - - - )
        OPR    =   YSTORE (Z12)
        OPL    =   OPR + OPL
        ZINDX (ZCD + ZB1)  =  ZINDX (ZCD + ZB1) + 1
        IF (ZINDX (ZCD + ZB1).LE.ZBONDS (ZBOUND + ZCD - 1))  GOTO 1Ø5
        Z1Ø    =   Z1Ø + 1
        YTEMP (Z1Ø + Z11)   =   OPL
        ZSAVE  =  1
           .
           .                             as for EXAMPLE 8.1.3
           .
        Z8    =   Z8 + 1
        IF ( Z8.EQ.ZCD) Z8  =  Z8 - 1
        GOTO 1Ø8
1Ø9     Z8    =   ZDIMS - 1
        GOTO 1Ø5
11Ø     ZPT   =  ZPT - 1
        CALL REDBND (ZCD)
        MARKER  =  -5
1Ø2     Z14      =  Ø
        Z11      =  ZYY (ZYYPTR)
111     Z14      =  Z14 + 1
```

The code YTEMP (Z11 + Z14) is then stored to be used as operand for the next operator.

Subroutines BDINFO, BNDSET and REDBND have the following functions:

1. BDINFO provides the number of dimensions, ZDIMS, and the pointer ZBOUND for the bound values of the non-scalar with NAMES index given by the first parameter.

2. BNDSET updates the bound information in ZCBNDS using the appropriate entries in ZBONDS

3. REDBND calculates the appropriate ZCBNDS entries for a reduction of 1 in the number of dimensions.

The value ZCOORD (ZCDPTR) gives the co-ordinate along which the reduction is to be applied.

For the above example, the first parameter of the FIND call is non-zero. Thus, if A is replaceable, the call of ELPERM in the optimised code is necessary to re-order the subscript values appropriately.

If, after the initial information is taken into account, it is known that A is a vector, then by a preliminary analysis of the generated code, it can be reduced immediately to the form given below.

```
         ZB1    =   ZPOINT (ZPT)
                .
                .
                .
         YTEMP (Z3 + Z4)   =   OPL
1Ø2      Z14    =   Ø
         Z11    =   ZYY (ZYYPTR)
111      Z14    =   Z14 + 1
```

Thus a considerable reduction of the generated code is possible. The code can be further reduced to the form:

```
         Z(1,1)   =   1
         OPL   =   < identity element for  +>
1ØØ      CALL ELPERM (------)
         ZPROD    =   Z(1,1)
         CALL ZADDR (A_NAMES,ZST,ZNUM,ZBOUND)
         Z2    =   ZST + ZPROD - 1
         OPR   =   OPR + OPL
         Z(1,1)   =   Z(1,1) + 1
         IF (Z(1,1).LE.ZBONDS (ZBOUND))  GOTO 1ØØ
         MARKER  =  Ø
             .
             .
             .
         < as above >
             .
             .
             .
```

If A is an n-dimensional array, the code can also be reduced by a preliminary analysis. Further reduction is then possible, but there is no advantage in altering the code so that the subscript values are stored in Z rather than ZINDX. A looping of the subscripts is required anyway to produce each result element. In this case, the reduced code would be of the form outlined below.

```
1Ø1      ZB1   =   ZPOINT (ZPT)
          .
          .
          .
         OPL   =   < identity element for  +>
         CALL ZADDR (A_NAMES,ZST,ZNUM,ZBOUND)
         Z_m   =   ZST - 1
1Ø5      Z_n   =   Z1Ø + 1
         Z12   =   Z_m + ZPERM (1,Ø,Ø, A_NAMES,Z_n )
```

```
          OPR  =  YSTORE (Z12)
                 '
                 '
                 '
111       Z14  =  Z14 + 1
```

Here ZPERM is a function which uses the subscript bounds and values to reorder the subscripts according to the first parameter value. Then for position Z_n , the actual position relative to the base address is produced as result. This simply involves applying the array mapping to the re-ordered subscript values.

The method outlined in §8.2 may be used to bring about the above transformations also.

8.2 Method of Optimisation

For a particular level of complexity, the reduction can be carried out independently of all other levels. However, since nesting of levels is allowed, the current state on entry to the higher level must always be stored. Thus, return can always be made to the correct state on exit from the higher level, and no vital information need be lost.

The method is illustrated using a finite state automaton. The state diagram for this automaton, together with the action required for each state, is given in Appendix 1Ø. At any time, the current state value and the statement encountered are used to provide the next state value.

The state value is initially zero and is updated according to the statement types encountered during the scan of the generated code. The statement types to be recognised during the scan are listed in Table 1Ø(a). Each statement type has an associated letter which is used for ease of reference in the state diagram.

During the scan of the generated code, entries are set up in several tables, namely:

1. DELETE
2. NSTATE
3. IENTRY
4. CODE

The table DELETE is two-dimensional and is used in the following way: It is not always possible to determine immediately whether particular lines of code may be deleted or not. For instance, in EXAMPLE 8.1.3, it is only when the last "FIND" call has been recognised that it is known whether the looping instructions are required. The DELETE table is updated during the scan as indicated below.

There is one entry in DELETE for each line of generated code. For a particular row, the columns have the following significance.

The first column gives the level of complexity. This is zero at the start and is updated according to the following criterion.

(a) If a statement of the form

```
ZB<integer>  =  ZPOINT (ZPT)
```

is recognised, then the level number (LEVLNO) is increased by 1.

(b) If a statement of the form

```
ZPT  =  ZPT - 1
```

is encountered, then the level number is decreased by 1.

The second column has value

(a) 1 for a line definitely to be deleted

(b) Ø for a line possibly to be deleted

(c) -1 for a line definitely not to be deleted.

In general, a number of DELETE entries with the second column Ø will be created, and these entries will be updated to 1 or -1 as the scan proceeds.

Thus, at the end of the scan, DELETE will have entries with value 1 or -1 in the second column, and the relevant lines can all be deleted at the same time.

Replacement of existing lines must also be considered. Again, it is not known immediately whether replacements will hav e to be made or not. For this reason, the table CODE is maintained. When it is known that a replacement line (or lines) may have to be produced, the appropriate line(s) is/are produced, and an entry is set up in a two-dimensional table CODE. Each entry in CODE consists of four parts, having the following significance:

(a) the first column gives the line number before which the insertion has to be made. (A replacement is considered to be a deletion followed by an insertion.)

(b) the second column gives the number of lines to be inserted

(c) the third column gives an indication of whether or not the replacement or insertion has to be made. This column (the insert entry) has value

(i) -1 initially

(ii) Ø if the insertion is not to be made

(iii) 1 if the insertion is to be made.

Thus -1 entries will be changed to either Ø or 1 as the scan proceeds and more information becomes available.

(d) the fourth column gives a pointer to the actual code to be used in the replacement or insertion.

A one-dimensional table NSTATE is also maintained. There is one row for every level of complexity. Each NSTATE entry records the current state value on entry to a higher level of complexity. Thus the scan of the generated code can be resumed correctly on return to the previous level of complexity. When a level has been scanned to completion, the NSTATE entry for that level should be set to -1 .

The stack IENTRY is used in updating the DELETE table. IENTRY has pointer IENPTR, which is Ø initially, and is updated as required. IENTRY (IENPTR) is Ø initially and is updated to 1 or -1, depending on the types of statement encountered during the scan.

The tables DELETE, CODE and NSTATE have pointers DELPTR, ICDPTR and NSTPTR, respectively. Each of these pointers has value Ø initially and is updated as required.

The variable LEVLNO gives the level number at any point, while ISTATE provides the current state value (within a particular level).

When label numbers are introduced in producing replacement lines of code, care should be taken to avoid duplicating existing line numbers. Similarly, if variable names Z <integer> are introduced, they must not conflict with existing variable names.

In implementing the finite state automaton, several values require to be stacked at intervals. Reference is made to these values in Appendix 1Ø.

Using this method, no actual replacements or deletions are made during the scan of the generated code. The tables DELETE and CODE are later used to produce the optimised code. On completion of the scan of the generated code, the array CODE should be ordered increasingly according to the values of the first column entries. Then DELETE and CODE can be scanned together and the necessary alterations made to the generated code.

For any type of statement not listed in Table 1Ø(a), the action required is independent of the state value. It is:

1. set DELPTR to DELPTR + 1
2. set DELETE (DELPTR,1) to LEVLNO
3. set DELETE (DELPTR,2) to -1 .

If, at any state, the next statement type is not given in the state diagram (see Diagram 1Ø(b)), then steps 1 to 3 above should be carried out.

Initially, the variables NSTPTR, LEVLNO, ISTATE, IENPTR and DELPTR all have value Ø .

8.3 Other Means of Optimising the Generated Code

In general, the generated code will be very inefficient. This can be seen by examining the sample translations in Appendix 9. Usually, however, it is a fairly simple matter to correct the inefficiencies.

Some means of optimising the generated code are discussed in §8.3.1 , while § 8.3.2 contains listings of optimised versions of the subroutines in Appendix 9.

8.3.1 Discussion of common types of inefficiency and their correction

The types of statement discussed here are:

1. COMMON statements
2. FORMAT statements
3. CONTINUE statements
4. Statements of the form Z <integer > = Ø followed by no other reference to Z <integer>
5. Statements to reset the values of MARKER and ZCPTR
6. Statements to reset the values of MARKER, ZCPTR, ZYPTR and ZYYPTR

7. Statements produced corresponding to an APL specification statement
8. Statements containing unnecessary variable names or redundant brackets
9. Statements corresponding to ⟶ statements.

1. When a function or subroutine header statement is translated, it is not known which global variables will be referred to in the function or subroutine body. Thus, at this stage, a complete list of COMMON statements must be generated, together with type specification statements for all the global non-scalars. If these are not referred to in the subsequent code, the non-executable statements may be removed. Several global parameters may be removed also from function or subroutine header statements.

If this type of optimisation is carried out, then the lowest level subroutine or function should be reduced first, then the next lowest level, and so on. This ensures that no unnecessary global parameters are thought to be necessary. For example, consider the following code:

```
SUBROUTINE A (Y1,Y2,Y3,Y4)
     B  =  Y1 + Y2
     CALL C(B,Y3,Y4)
     RETURN
     END
SUBROUTINE C (X1,X2,X3)
     D  =  3*X1
     RETURN
     END
```

Here, if A is reduced first, Y3 and Y4 are thought to be necessary, which is an incorrect assumption.

2. A number of FORMAT statements are produced at the end of each function or subroutine decoded. Those unnecessary may easily be removed.

3. At the start of each section of code corresponding to an APL line, a statement of the form

```
        <label>   CONTINUE
```

is generated. This allows for statements of the form

```
        → <expression>
```

which may occur later. However, in certain cases, <expression> may only have a finite range of values. For example, consider the statement

```
        → 2 X N > Ø
```

Here < expression > can only have values Ø or 2.

It may be possible to eliminate certain statements of the form

```
        < label > CONTINUE
```

see the examples given in § 8.3.2 .

4. If square brackets occur on a line, then a statement of the form

```
        Z<integer > =  Ø
```

is generated. This is done in order that Z < integer > may be used as a counting variable if the subscript value is non-scalar. (For example, consider the variable Z1 in EXAMPLE 8.1.7 .)

If the subscript value is scalar, the statement Z <integer> = Ø is redundant, and may be removed.

A statement of this type may easily be detected, as there will be no further reference to Z <integer >.

5. After each statement of the form

```
     <label > CONTINUE
```

the statements

```
     MARKER  =  Ø

     ZCPTR   =  Ø
```

are also generated. These reset the type of the "expression-so-far" back to scalar. They are necessary in cases where MARKER has been set to some other type value. Consider the code given below:

```
    < n > CONTINUE
          MARKER  =  Ø
          ZCPTR   =  Ø
             :
             :
             :
  < n+1 > CONTINUE
          MARKER  =  Ø
          ZCPTR   =  Ø
             .
```

If there are no subroutine or function calls between < n > CONTINUE and < n+1 > CONTINUE , and the value of MARKER has not been explicitly changed, then the second two statements

```
          MARKER  =  Ø          (a)
          ZCPTR   =  Ø          (b)
```

are redundant, and may be removed.

If the above conditions are satisfied, except that there is a call of the subroutine OUT2 between the statements labelled <n> and < n+1 >, then the second two statements (a) and (b) are still redundant. OUT2 (listed in module library SARUN) does not alter the value of MARKER. In general, checks may be made for unnecessary individual resetting of these variables.

6. If two sequences of statements

```
MARKER  =  Ø
ZCPTR   =  Ø
ZYPTR   =  1
ZYYPTR  =  1
```

occur, then the second sequence may be redundant. If any subroutine is called which updates the above variables, then the statements will be necessary. If no such subroutine occurs, and the values have not been explicitly altered, then the second sequence is redundant and may be removed. In certain cases, only some of the above statements should be removed.

7. For a specification statement of the form

$A \longleftarrow B$

where B is a non-scalar, code of the form

```
<initiallisation statements>
CALL FIND1 (--- B_NAMES --- Y_n --- )
Y_n+1  =  Y_n
CALL SPECS (- - - )
<code to complete loops >
```

is generated.

If initiallisation statements have not first been set up, it is necessary to test the value of MARKER to determine whether an explicit assignment may be made, or whether a call of SPECS must be generated.

In cases where MARKER has previously been set to a specific value, the test on its value may be removed. For example, consider subroutine SPHERE listed in Appendix 9. SPHERE contains the statements

```
          MARKER  =  Ø
          ZCPTR   =  Ø
          Y2   =   4 * (3.14159*(R * R))
          IF (MARKER.EQ.-5.OR.MARKER.EQ.-3)  GOTO 111
          SURF  =  Y2
          GOTO 117
111       CALL SPECS (- - - -)
          CALL SPECB
117       CONTINUE
```

Here the specification part may be reduced to

```
          Y2   =   4 * (3.1459*(R * R))
          SURF   =   Y2
```

8. The above code may be further reduced to

```
          SURF   =   4 * 3.14159 * R * R
```

by eliminating the unnecessary variable Y2 and removing some redundant brackets. Care must be taken when removing brackets to ensure that they are in fact redundant.

9. To allow for statements of the form

```
          ⟶ < expression >
```

where the value of < expression > is not known until execution time of the converted routine, statements of the form

```
          Z1  =  < expression >
          GOTO 1ØØØ
```

are generated (see Chapter 6).

Correspondingly at the end of each subroutine or function, the statements

```
1ØØØ      IF (Z1.LE.Ø.OR.Z1.GT.n)  CONTINUE
          GOTO (1,2,3,---,n)  Z1
```

are generated, where n is the number of lines in the function.

The above two statements will be unnecessary if no → statements are present in the APL subroutine or function.

8.3.2 Optimised versions of the subroutines SPHERE, BASE and CI (Listed in Appendix 9)

1. SPHERE may be reduced to the form given below.

```
          SUBROUTINE SPHERE
          IMPLICIT REAL (A-Y)
          IMPLICIT INTEGER (Z-Z)
C*****    THE NEXT 3 STATEMENTS WERE INSERTED BY HAND
          COMMON / CD1 / R
          COMMON / CD2 / SURF
          COMMON / CD3 / VOL
          SURF   =   4 * 3.14159 * R * R
          VOL    =   SURF * R/3
          CALL LOCREM
          RETURN
          END
```

The subroutine LOCREM is called to remove entries from NAMES corresponding to local variable names when these are no longer required. It will have no effect in the above case, but should be called so that the appropriate pointer variables will be updated (see the version of LOCREM in module library SARUN).

In the next examples, the global non-scalars are removed from subroutine calls and these are assumed to be in COMMON within the subroutines. A comparison of this method with the original (see Appendix 9) is given at the start of Chapter VIII.

2. Subroutine CI (for compound interest calculation) may be reduced to the form given below.

```
          SUBROUTINE  CI
          IMPLICIT REAL (A-Y)
          IMPLICIT INTEGER (Z-Z)
          COMMON / C351 / MARKER
          COMMON / C916 / ZSAVE
          COMMON / C726 / ZLIMZ6
          CALL OUT2 (15,-1)
          MARKER  =  Ø
          ZCPTR   =  Ø
          READ (5,1Ø2) ZVBND
               .
               .
               .
128       CONTINUE
          CALL OUT2 (54,-1)
          READ (5,1Ø2) ZVBND
               .
               .
               .
151       CONTINUE
          CALL OUT2 (87,-1)
          READ (5,1Ø2) ZVBND
               .
               .
               .
```

```
174       CONTINUE
          CALL OUT2 (116,-1)
               ,
               ,
               ,
186       CONTINUE
1Ø1       FORMAT (1X,G12.6)
1Ø2       FORMAT (G12)
1Ø4       FORMAT (1Ø G12.6)
1Ø6       FORMAT (1X,/)
          CALL LOCREM
          RETURN
          END
```

All the subroutines called should be examined for generality. If, for instance, A handles cases 1, 2, 3 and 4, and it is known that case 3, say, will be applied, then A could be replaced by a simpler function, thus increasing efficiency.

3. Subroutine BASE produces the representation of a number B to the base N. Assuming full information is available for YZ, an array in which the result is accumulated, the following reduction in code is possible.

```
          SUBROUTINE BASE (B,N)
          IMPLICIT REAL (A-Y)
          IMPLICIT INTEGER (Z-Z)
          COMMON /C351/ MARKER
          COMMON /C916/ ZSAVE
```

```
  1       CONTINUE
          Y2  =  Ø
          IF (Y2.NE.Ø)  GOTO 111
          MARKER  =  -3
          ZCPTR   =  Ø
          GOTO 112
111       - - - - - -
             '
             '
             '
  2       CONTINUE
          Y7  =  B
          Y9  =  N
          IF (Y7.NE.Ø) Y9 = Y9 - ABS (Y7)*AINT (N/ABS(Y7))
          R   =  Y9
  3       CONTINUE
          MARKER  =  Ø
          ZCPTR   =  Ø
          ZSAVE   =  Ø
          CALL ZADDR (1,ZST,ZNUM,ZBOUND)
          Z1Ø2  =  ZST - 1
          Z1ØØ  =  1
          CALL ZADDR (1,ZST,ZNUM,ZBOUND)
          Z1Ø1  =  ZST - 1
138       Z1Ø1  =  Z1Ø1 + 1
          Y17   =  LPERM (Z1Ø1,14,Ø,R,1)
          Y18   =  Y17
          Z1Ø2  =  Z1Ø2 + 1
          CALL SPECA (Z1Ø2,Y12)
          Z1ØØ  =  Z1ØØ + 1
          IF (Z1ØØ.LE.ZNUM)  GOTO 138
          CALL SPECB
```

Further optimisation is obviously possible here. This is an example showing how full generality must be catered for in code generation, but that some calls may be eradicated by analysis of the code.

```
147       CONTINUE

          Y2Ø  =  N/B

          IF (Y2Ø.GE.Ø)  GOTO 153

          Y2Ø  =  ABS(Y21 - (1-1E-8))

153       Y2Ø  =  AINT (Y2Ø)

          N  =  Y2Ø

          CALL OUT2 (1,1)

          Z23  =  Ø
            .
            .
            .
172       CONTINUE

1ØØØ      IF (Z1.LE.Ø.OR.Z1.GT.2)  CONTINUE

          GOTO (1,2), Z1

          CALL LOCREM

          RETURN

          END
```

In this example, Z1 always has value ≤ 2 . Thus, if some analysis of the code is carried out, the statements starting with that labelled 1ØØØ can be updated as shown.

No account has been made of the fact that the SPECS and FIND1 calls refer to the same non-scalar. Further optimisation is possible here.

CONCLUSION

It has been shown that it is possible to obtain a translation of an APL routine into another high-level language. It cannot be denied that the translated code is inefficient, although it is possible to bring about a considerable increase in efficiency.

One major factor affecting the efficiency of the generated code is the possible presence of unnecessary "FIND" calls. These can be removed by a method based on recognition of specific statement types, as discussed in Chapter VIII, §8.1 and §8.2, and in Appendix 10. The removal of "FIND" calls involves little or no analysis of the generated code. However, in cases where any one of a number of different paths may be taken (here each case must be considered and the generated code soon becomes unwieldy), it is often possible to eliminate all but one of the paths. Such forms of optimisation involve analysis of the generated code, often making use of additional information for non-scalars which was not taken into account at code generation time.

The inefficiency of the generated code is excusable, since no attempt whatsoever was made to optimise code at the time of its generation. If the two methods outlined above, that is

1. replacement of unnecessary "FIND" calls,
2. analysis of the generated code,

are combined, then the increase in efficiency can be very great.

In retrospect, it is thought that even greater efficiency may be obtained by a pre-analysis of the APL code to be translated. APL conceals a great many operations (such as the testing of variable types) which must be carried out explicitly in other high-level languages. For this reason the generated code must make provision for a large number of possibilities. It is probable that the number of cases to be handled could be reduced by restating the APL problem.

REFERENCES

1. F.R.A. Hopgood : "Compiling Techniques", 1969, MacDonald Computer Monographs 8

2. D. Knuth : "Art of Computer Programming", (Volume 3), Addison-Wesley, April, 1973

3. V.L. Moruzzi : "APL/FORTRAN Translations", December, 1971

4. B. Randell, L.J. Russell : "ALGOL 60 Implementation", Academic Press, London

5. M.M. Sayers : "APL to FORTRAN", Paper presented at SEAS Conference in Gothenburg, Sweden, in September, 1972

6. K.W. Smillie : "APL AND STATISTICS . PROGRAMS OR INSIGHT?" Department of Computing Science, University'of Alberta, Edmonton, Alberta, Canada.

7. H. Van Hedel : "An APL Batch Processor", Applied Mathematical Division, European Space Technology Center, Noordwijk, Holland.

APPENDIX 1

SYMBOL	REPRESENTATION	SYMBOL	REPRESENTATION
/	/	-	-
\	# C	×	# F
←	# D	÷	# G
∧	# J	*	*
∨	# K	⌈	# H
<	<	⌊	# I
≤	# L	|	|
=	=	⍳	# P
≥	# M	⍴	# Q
>	>	,	,
≠	# N	!	!
∊	# O	⌽	# R
⊤	# S	○	# U
⊥	# T	?	?
↑	# V	⍉	# X
↓	# W	⍟	@ C
⍲	@ D	⊖	@ I
⍱	@ E	→	# E
⌿	@ J	~	—•
⍀	@ K	⊣	# Y
+	+	⍏	@ G
⍖	@ H	⍙	@ P
[	# A	.	.
]	# B	(overbar)	@ L
(	(	:	:
)	)	∇	@ M
;	;	⍫	@ N

SYMBOL	REPRESENTATION	SYMBOL	REPRESENTATION
∘	# Z	A-Z	A-Z
⎕	@ A	0-9	0-9
⍞	@ B	$\underline{\mathrm{A}}$-$\underline{\mathrm{Z}}$	_A - _Z
A	@ F	'	'
∆	@ O		

APPENDIX 2

TABLE OF USEFUL INFORMATION FOR APL SYMBOLS

	SYMBOL	ADDRESS IN SYMBOL TABLE	Z-CODE VALUE	MACRO NUMBER
P	/	1	19	1
U	\	2	20	2
R	←	3	21	3
E	∧	4	34	4
L	∨	5	35	5
Y	<	6	36	6
D	≤	7	37	7
Y	=	8	38	8
A D	≥	9	39	9
I	>	10	40	10
C	≠	11	41	11
O	∊	12	43	12
P	⊤	13	50	13
E	⊥	14	51	14
R	↑	15	55	15
A	↓	16	56	16
T	⍲	17	68	17
O	⍱	18	69	18
R	⌿	19	74	19
S	⍀	20	75	20

	SYMBOL	ADDRESS IN SYMBOL TABLE	Z-CODE VALUE	MACRO NUMBER
				D M
BOTH MONADIC AND DYADIC OPERATORS	+	21	26	21 , 39
	-	22	27	22 , 40
	×	23	28	23 , 41
	÷	24	29	24, 42
	*	25	30	25 , 43
	⌈	26	31	26 , 44
	⌊	27	32	27 , 45
	|	28	33	28 , 46
	⍳	29	44	29 , 47
	ρ	30	45	30 , 48
	,	31	47	31 , 49
	!	32	48	32 , 50
	⌽	33	49	33 , 51
	○	34	52	34 , 52
	?	35	53	35 , 53
	⍉	36	62	36 , 54
	⍟	37	67	37 , 55
	⊖	38	73	38 , 56

D - DYADIC

M - MONADIC

	SYMBOL	ADDRESS IN SYMBOL TABLE	Z-CODE VALUE	MACRO NUMBER
MONADIC OPERATORS AND SPECIAL SYMBOLS	→	39	22	57
	↝	40	54	58
	⌶	41	63	59
	⍏	42	71	60
	⍒	43	72	61
	[	44	14	62
	]	45	15	63
	(	46	16	64
	)	47	17	65
	;	48	18	66
	∘	49	64	67
	⎕	50	65	68
	⍞	51	66	69
	⍝	52	70	70
	A-Z	53 - 78	86 - 111	
	Δ	79	112	
	$\underline{A}$-$\underline{Z}$	80 - 105	113 - 138	
	⍙	106	139	
	0-9	107 - 116	140 - 149	71 INNER PRODUCT
	.	117	150	
	overbar	118	151	
	blank	119	152	72 HET. OUTPUT
	'	120	153	
	:	121	154	73
	∇	122	155	74
	⍫	123	160	75

APPENDIX 3

MACRO INSTRUCTIONS AND THEIR FUNCTIONS

In the following table the form < ? n > is used to represent the value of IDSTK (IDLPTR + n).

For macro instructions FL, PC, SL and V, it has been assumed that IND = 1 and IDOLR = 1ØØ initially, so that

variables Z1, Z2, ---- are used

and label numbers 1ØØ, 1Ø1, ---- are used.

MACRO INSTRUCTION	FUNCTION
% --- %	COPIES THE TEXT --- TO MTEMP, STARTING FROM POSITION TEMPR. A CONTINUATION LINE IS CREATED IF NECESSARY.
&	TRANSFERS THE CONTENTS OF MTEMP TO THE OUTPUT MEDIUM (USER-DEFINED)
+B	INCREMENTS THE POINTER IBIT BY 1
+D	INCREMENTS THE VARIABLE IND BY 1
+I	INCREMENTS THE VARIABLE IDOLR BY 1
+N	INCREMENTS THE VALUE OF NUMBER (NMBR) BY 1
< label> : --- and = <label>	USED TO PRODUCE UNCONDITIONAL BRANCH TO MACRO STATEMENT LABELLED < label >
S<integer>,<expression>	STORES THE VALUE expression IN SS(<integer>), USED FOR COUNTING IN LOOPS
& --- &	CONCATENATES --- TO THE LAST ENTRY IN ITEMP, THUS CREATING A LONGER ENTRY

MACRO INSTRUCTION	FUNCTION
& <integer>	DECREASES SS(<integer>) BY 1. IF SS(<integer>) IS Ø, SEQUENTIAL EXECUTION CONTINUES; OTHERWISE BRANCH TO START OF LOOP AND REPEAT
£<expression>	EVALUATES AN EXPRESSION (FOR EXAMPLE, £ -1) AND USES THE VALUE AS A LABEL NUMBER, WHICH IS STORED ON MTEMP. THE ONLY OPERATORS ALLOWED IN <expression> ARE + AND - . OPERANDS MAY BE INTEGERS, S <integer> or £ . THE VALUES OF SS(<integer>) AND IDOLR ARE SUBSTITUTED FOR S<integer> AND £ RESPECTIVELY.
?<expression>	EVALUATES <expression> AND STORES THE VALUE OF IDSTK (IDLPTR + <expression>) ON MTEMP. EXPRESSIONS MAY INVOLVE OPERATORS +, -, *, /, **, (AND). BRACKETS MAY BE NESTED TO A MAXIMUM OF 1Ø LEVELS DEEP. OPERANDS MAY BE INTEGERS, S<integer>, IND OR INE. ?'s MAY BE NESTED TO TWO LEVELS DEEP
AO	GENERATES THE CODE .AND. (FOR ∧ OR ⊽) .OR. (FOR V OR ⊼)
BB	USES A SUBSCRIPT COUNT TO PRODUCE THE NUMBER OF THE SUBSCRIPT BEING REFERENCED IN MTEMP
BP	RESETS THE POINTER, IBPTR, FOR THE ARRAY ITEMP. IBPTR ← ITEMP (IBPTR)
BR	GENERATES AN OPENING ROUND BRACKET ON MTEMP (TEMPR) AND INCREMENTS TEMPR BY 1
CB	GENERATES A CLOSING ROUND BRACKET ON MTEMP (TEMPR) AND INCREMENTS TEMPR BY 1
CS	GENERATES NON-EXECUTABLE STATEMENTS TO BE PLACED AFTER A FUNCTION OR SUBROUTINE HEADER STATEMENT (SEE SUBROUTINE CSREC IN MODULE LIBRARY SALIB)
DB	GENERATES CODE OF THE FORM ZBA = Ø CALL DREC (<?1>,ZBA,ZCBNDS,ZCPTR,ZBOUNDS, ZSTORE)

MACRO INSTRUCTION	FUNCTION
DS	IF VARIABLE ZIND ≠ ∅, NO EFFECT. OTHERWISE, GENERATES CODE OF THE FORM MARKER = -5 <label 1> Z <integer> = ∅ <label 2> Z <integer> = Z <integer> + 1 <integer> and <label 2> ARE STORED IN ZIND AND ZNO RESPECTIVELY
DS2	IF VARIABLE ZIND ≠ ∅, NO EFFECT. OTHERWISE, GENERATES CODE OF THE FORM < label 1> Z <integer> = ∅ < label 2 > Z <integer> = Z <integer> + 1 <integer> and <label 2 > ARE STORED IN ZIND AND ZNO RESPECTIVELY
FA	GENERATES CODE FOR A FUNCTION CALL ON MTEMP, STARTING FROM POSITION TEMPR. (GLOBAL VARIABLES ARE INSERTED SEPARATELY AT THE END, USING THE --- INSTRUCTION.)
FL	GENERATES CODE OF THE FORM ZSAVE = 1 ZINDX (Z<ISAVE>) = ZINDX (Z<ISAVE>) + 1 IF (ZINDX (Z<ISAVE>).LE.ZCBNDS (ZCPTR)) GOTO < JSAVE > Z2 = Z<KSAVE > + 1 1∅1 ZINDX (Z2) = 1 Z2 = Z2 + 1 IF (Z2.LE.(ZB<integer> + ZCPTR)) GOTO 1∅1 1∅2 ZINDX (ZB<integer> + Z<KSAVE>) = ZINDX (ZB<integer> + Z<KSAVE>) + 1 IF (ZINDX(ZB<integer> + Z<KSAVE>).LE. ZBOUNDS (Z<KSAVE>)) GOTO 1∅3 IF (Z<KSAVE>.EQ.1) GOTO 1∅4 ZINDX (ZB<integer> + Z<KSAVE>) = 1 Z<KSAVE> = Z<KSAVE> - 1 * GOTO 1∅2 1∅3 Z<KSAVE> = Z<KSAVE>- 4 - 1 GOTO <JSAVE > 1∅4 ZPT = ZPT - 1 THE VALUES < ISAVE>, <JSAVE > AND < KSAVE > HAVE PREVIOUSLY BEEN STORED USING INSTRUCTION SL. * FOR REDUCTION, THE CODE IF (Z<KSAVE>.EQ.ZCD) Z<KSAVE> = Z<KSAVE> - 1 IS GENERATED HERE

MACRO INSTRUCTION	FUNCTION
FN <integer>	GENERATES CODE FOR AN IDENTIFIER ON MTEMP. THE IDENTIFIER HAS INDEX IDSTK (IDLPTR + <integer>) IN NAMES
FP	INCREASES IFNPTR VALUE BY 1 AND SETS IFIND (IFNPTR) TO 1-14, DEPENDING ON THE OPERATOR CURRENTLY BEING HANDLED
FS	SETS IFIND (IFNPTR) TO Ø AND DECREMENTS IFNPTR BY 1
FV	USED IN PRODUCTION OF "FIND" CALLS TO GENERATE THE FIRST 3 PARAMETER VALUES ON MTEMP, STARTING FROM POSITION TEMPR. FOR EXAMPLE I, J, K
FV2	USED IN PRODUCTION OF "FIND" CALLS TO GENERATE THE SECOND AND THIRD PARAMETERS ON MTEMP, STARTING FROM POSITION TEMPR. FOR EXAMPLE ,J,K (USED IN REDUCTION MACRO, WHERE FIRST PARAMETER ALWAYS HAS VALUE 1).
FX	USED IN PRODUCTION OF FUNCTION CALLS FOR NON-SCALAR OPERANDS. IF OPERAND IS NOT A PARAMETER, ITS NAMES INDEX IS GENERATED ON MTEMP, STARTING FROM POSITION TEMPR. OTHERWISE, THE FORM 'ZF1' OR 'ZF2' IS PRODUCED ON MTEMP, DEPENDING ON THE POSITION OF THE OPERAND IN THE PARAMETER LIST.
F <integer>	USED TO PRODUCE FUNCTION CALLS FOR DIFFERENT TYPES AND NUMBERS OF PARAMETERS. FOR EXAMPLE, F1 GENERATES FN.NAME (Z<integer> F5 GENERATES ,ZF2, THUS F1 F5 %---)% WOULD GENERATE A CALL FOR A SCALAR LEFT PARAMETER AND A NON-SCALAR RIGHT PARAMETER
FØ	GENERATES CODE OF THE FORM <FUNCTION NAME> = <RESULT VARIABLE> WRITE (6,1ØØ) FUNCTION NAME
ID	GENERATES THE VALUE OF THE IDENTITY ELEMENT FOR AN OPERATOR ON MTEMP, STARTING FROM POSITION TEMPR.

MACRO INSTRUCTION	FUNCTION
IF < a > < rel.op> m n	CREATES A CONDITIONAL BRANCH TO LABEL NUMBER n IN THE MACRO BODY. BRANCH IF TEST SATISFIED; OTHERWISE SEQUENTIAL EXECUTION. <a> MAY BE (i) ?<expression> (ii) A (TESTS FOR LOOPS SET UP FOR NON-SCALARS) (iii) F (TESTS THE VALUE OF IFCN) (iv) I (TESTS THE VALUE OF IEXP) (v) MR (TESTS THE VALUE OF MARK) (vi) N (TESTS THE VALUE OF FNI) (vii) N < expression > (TESTS THE VALUE OF NAMES (?<expression>)) (viii) O (TESTS THE VALUE OF IDSTK (IoPTR)) (ix) S <integer > (TESTS THE VALUE OF SS(<integer>) (x) X (TESTS THE VALUE OF NAMES (?1 + 2) (xi) Y (TESTS THE 1st CHARACTER OF THE LAST ENTRY IN MTEMP) (xii) Z (TESTS THE VALUE OF ZMARK) THE ABOVE SYMBOLS REPRESENT THE QUANTITIES SHOWN IN BRACKETS rel.op. MAY BE EQ,NE,GE,GT,LE,LT m IS A POSITIVE OR NEGATIVE INTEGER
IF < a > <rel.op> m **	CONDITIONAL STOP. AS ABOVE, BUT a MAY ONLY BE (i), (v) OR (xii)
IND<expression>	REPRESENTS MACRO EXPRESSICN STARTING WITH IND.VALUE (AN INTEGER) IS CALCULATED AND STORED ON MTEMP, STARTING FROM POSITION TEMPR
INE<expression>	REPRESENTS MACRO EXPRESSION STARTING WITH INE. VALUE (AN INTEGER) IS PLACED ON MTEMP, STARTING FROM POSITION TEMPR
LI	TRANSFERS A COMMENT FROM NAMES TO MTEMP AND SETS THE NAMES ENTRY TO -1's (FOR POSSIBLE GARBAGE COLLECTION)
LM	GENERATES AN INTEGER (1-6) ON MTEMP, STARTING FROM POSITION TEMPR. (USED TO DISTINGUISH RELATIONS IN RELATIONAL OPERATOR MACRO)

MACRO INSTRUCTION	FUNCTION
LN	AS FOR LM, BUT VALUES 6-1 ARE GENERATED
LO	TRANSFERS THE LEFT OPERAND FROM ITEMP TO MTEMP. NLEFT IS SET SO THAT ENTRY WILL BE REMOVED FROM ITEMP LATER.
LO+	AS FOR LO, BUT NLEFT SET SO THAT ENTRY WILL NOT BE REMOVED FROM ITEMP
L1	CHECKS FOR ENTRY IN COLUMN 1 OF LTABLE WITH VALUE <?1> . WHEN ENTRY IS FOUND, THE CORRESPONDING ENTRY IN COLUMN 2 OF LTABLE IS STORED ON MTEMP, STARTING FROM POSITION TEMPR.
MR<integer>	STORES THE VALUE OF <integer> IN MARK
MX - - -	STORES THE VALUE OF THE ENTIRE EXPRESSION ON MTEMP, STARTING FROM POSITION TEMPR. MX INDICATES AN EXPRESSION STARTING WITH IMAXFN (IMXPTR).
N	STORES THE VALUE OF NUMBER (NMBR) ON MTEMP, STARTING FROM POSITION TEMPR.
NB	INCREMENTS NMBR BY 1
O	GENERATES CODE + OR - OR * OR / OR ** ON MTEMP, STARTING FROM POSITION TEMPR. THE CODE DEPENDS ON THE CURRENT OPERATOR (+ OR - OR x OR ÷ OR *)
PC	GENERATES CODE OF THE FORM CALL BDNO (Z2,ZCPTR) IF (Z<A>.LT.Z2) GOTO <B> <A> IS GIVEN BY ZIND (SET USING DS INSTRUCTION) AND <B> BY ZN.O
PL	GENERATES CODE OF THE FORM Z <integer> = <left operand> FOR INTEGERS AND Y <integer> = <left operand> FOR REAL VARIABLES. <left operand> IS OBTAINED FROM ITEMP. NLEFT IS SET SO THAT ENTRY WILL BE REMOVED FROM ITEMP LATER.
PL+	AS FOR PL, BUT NLEFT IS SET SO THAT ENTRY WILL NOT BE REMOVED FROM ITEMP.
PR	AS FOR PL, BUT <right operand> IS USED INSTEAD

MACRO INSTRUCTION	FUNCTION
PR+	AS FOR PL+, BUT <right operand> IS USED INSTEAD
RCM	USED FOR INTERRUPTION OF A MACRO EXPANSION. THE VALUES OF IND AND IDOLR ARE STORED SO THEY MAY BE USED WHEN THE REMAINDER OF THE MACRO IS EXPANDED LATER. AN ADDRESS FOR RETURN TO THE MACRO BODY IS ALSO STORED.
RE	RESETS THE VALUE OF IDPTR TO IDLPTR
RL	GENERATES CODE .EQ. OR .NE. OR .LT. OR .LE. OR .GT. OR .GE. ON MTEMP, STARTING FROM POSITION TEMPR. THE CODE GENERATED DEPENDS ON THE CURRENT OPERATOR (= OR ≠ OR < OR ≤ OR > OR ≥)
RO	TRANSFERS THE RIGHT-MOST ENTRY FROM ITEMP TO MTEMP. IF NLEFT IS SET, 2 ENTRIES ARE REMOVED FROM JTEMP; OTHERWISE ONLY ONE ENTRY IS REMOVED.
RO+	AS FOR RO, BUT NO ENTRIES ARE REMOVED FROM ITEMP.
R1	TRANSFERS THE RIGHT-MOST ENTRY (WITH ENCLOSING BRACKETS, IF ANY, REMOVED) FROM ITEMP TO MTEMP.
R1+	AS FOR R1, BUT ENTRY IS NOT REMOVED FROM ITEMP
R2	REMOVES RIGHT-MOST ENTRY FROM ITEMP
S	TRANSFERS CONTENTS OF MTEMP TO ITEMP. MTEMP IS CLEARED.
S+	AS ABOVE, BUT MTEMP IS NOT CLEARED.
S <integer>, <expression>	STORES THE VALUE OF <expression> IN SS(<integer>)

SL	GENERATES CODE OF THE FORM
	ZB1 = ZPOINT (ZPT)
	ZPT = ZPT + 1
	CALL STARTS (<?1>,Z1,Z2,ZNC)
	ZPOINT (ZPT) = ZB1 + Z2
	Z3 = 1
	1Ø1 Z4 = ZB1 + Z3
	ZINDX (Z3) = 1
	Z3 = Z3 + 1
	IF (Z3.LE.Z2) GOTO 1Ø1
	Z5 = ZB1 + Z2
	Z6 = Z2 - 1
	ZSAVE = Ø
	<ISAVE> IS SET TO 5, <JSAVE> TO 1Ø1 AND <KSAVE> TO 6 TO BE USED TO GENERATE THE END OF THE LOOPS LATER

MACRO INSTRUCTION	FUNCTION
SL2	GENERATES CODE OF THE FORM CALL STARTS (<?1>,Z1,Z2,ZNC)
STK	INCREMENTS IDPTR BY 1, STORES IDLPTR VALUE ON IDSTK (IDPTR) AND SETS IDLPTR TO IDPTR.
T<integer>	STORES THE VALUE OF<integer>IN TEMPR
V	GENERATES CODE OF THE FORM CALL VSET (<?1> , Z2) VSET IS PRESENT IN MODULE LIBRARY SARUN
Z+	GENERATES CODE OF THE FORM Z <integer> = Z <integer> + 1 WHERE <integer> IS GIVEN BY THE VALUE OF IND.
ZA	GENERATES CODE OF THE FORM Z <integer> = 1 IND AND IDCLR ARE STORED IN ISUBS (N,1) AND ISUBS (N,2) RESPECTIVELY. (N IS THE TNDEX OF THE FIRST EMPTY ROW.) USED FOR SUBSCRIPT LOOPS.
ZB - - - -	REPRESENTS AN EXPRESSION STARTING WITH ZLAB. EXPRESSION IS EVALUATED AND THE VALUE (A LABEL NUMBER) IS STORED ON MTEMP, STARTING FROM POSITION TEMPR.
ZC	GENERATES THE VALUE OF ZNUM ON MTEMP, STARTING FROM POSITION TEMPR.
ZD	GENERATES CODE OF THE FORM Z<integer> = Ø AND STORES <integer> IN ZNA
ZE	PRODUCES THE VALUE OF ZNA ON MTEMP
ZI	GENERATES THE VALUE OF ZIND (OR IND, IF ZIND = Ø) ON MTEMP, STARTING FROM POSITION TEMPR.
ZM	GENERATES THE VALUE OF ZMARK ON MTEMP, STARTING FROM POSITION TEMPR.

ZP	IF ISBPTR = 0, NO EFFECT. OTHERWISE, GENERATES CODE OF THE FORM
	Z < A> = Z < A> + 1
	IF (Z< A> .LT.ZBONDS (Z< B>)) GOTO <C>
	ZPT = ZPT - 1
	WHERE < A> ≡ ISUBS (N,1)
	< B> ≡ ISUBS (N,1) - 1
	< C> ≡ ISUBS (N,2)
	AND N ≡ ISBPTR

MACRO INSTRUCTION	FUNCTION
ZS	GENERATES CODE OF THE FORM Z < integer > = ZYY (ZYY PTR - 1)
ZT	GENERATES CODE OF THE FORM Z< integer > = ZYY (ZYYPTR)
ZW - - -	REPRESENTS AN EXPRESSION STARTING WITH ZNUMB. THE EXPRESSION IS EVALUATED AND THE RESULT STORED IN MTEMP, STARTING FROM POSITION TEMPR.
ZY	GENERATES CODE OF THE FORM Z < integer > = ZY (ZYPTR - 1)
ZZ	GENERATES CODE OF THE FORM Z <integer> = ZY (ZYPTR)
Z< integer>	GENERATES CODE OF THE FORM Z < integer 1> = < integer >
#	STOP

APPENDIX 4

A list of all the macro bodies used in the APL-FORTRAN translation is included for reference purposes.

A header line has been included with each macro body. The header line gives the macro number, as well as a brief description of the contents or function of the macro body. The macro processor never accesses the header lines for the macro bodies, and thus the header lines could be omitted. Omission of the header lines would significantly reduce the amount of space required for the macro bodies. The header lines (together with the surrounding blank lines) have been inserted simply for readability of the macro code.

```
   PRODUCTION OF EVFIND CALL   (NO.1)

  IF A NE 0 1  SL  1: +D  £-1  %Y%  IND  %=EVFIND(%  FV  %,%  FX  %,ZPOINT,ZPT,Z
CBNDS,ZCPTR,YBOUND,ZINDX,ZSTORE,ZBONDS,ZCOORD,ZCOPTR)%  &  %Y%  IND  S  #

   PRODUCTION OF IRFIND CALL   (NO.2)

  %IF(Z%  IND  %.GT.ZLIM21) CALL GVOVER(21,&%  +I  £+5  %)%  &  %YBOND(Z%  IND
%)=%  RI  &  PC  £  ZT  %ZYYPTR=ZYYPTR+1%  &  %IF(ZYYPTR.GT.ZLIM8) CALL GVOVER(8
,&%  +I  £  %)%  &  SL  ZG  £-1  Z+  %YTEMP(Z%  ZC  %+Z%  IND  %)=IRFIND(%  FV
%,ZPOINT,ZPT,ZCBNDS,ZCPTR,YBOUND,ZINDX,ZSTORE,ZBONDS,ZCOORD,ZCOPTR,YBOND)%  &  %
ZYY(ZYYPTR)=Z%  IND  &  FL  £-7  ZS  %ZYY(ZYYPTR)=0%  &  %ZYYPTR=ZYYPTR-1%  &  D
S  %YTEMP(Z%  ZC  %+Z%  IND  %)%  S  #

   SPECIFICATION   (NO.3)

  IF ?1 LT 0 1  PR  &  IF A NE 0 3  %IF(MARKER.NE.0) GOTO %  +I  £  &  IF ?2 NE
0 30  FN1  %=Y%  IND  &  %GOTO %  £+6  &  £  3: %CALL SPECS(%  FX  %,Y%  IND  %,
Z%  ZI  %,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZSTORE,ZCOORD,ZCOPTR,YBOUN
D)%  &  PC  %CALL SPECB(YSTORE)%  &  +I  £  %CONTINUE%  &  #  1: LO  %=%  RI+  R
2  #  30: SL  =3

   AND,OR   (NOS.4-5)

  PL  &  PR  &  +D  %Z%  IND  %=0%  &  %IF(Z%  IND-1  %.EQ.1%  AO  %Z%  IND-2  %
.EQ.1)Z%  IND  %=1%  &  %Z%  IND  S  #

   EQ,NE,LT,LE,GT,GE   (NOS.6-11)

  IF ?1 LT 0 1  IF ?2 NE 0 2  IF ?3 LT 0 3  ZO  %IF(%  FN1  RL  FN3  %)Z%  IND
%=1%  &  4: %Z%  IND  S  #  3: PR  &  ZO  %IF(%  FN1  RL  %Y%  IND-1  %)Z%  IND
%=1%  &  =4  2: IF ?3 LT 0 5  +D  %Z%  IND  %=IRELOP(%  FX  %,%  ?3  %,%  LM  %
,YBOUND,ZBONDS,YSTORE,ZSTORE,XBOUND)%  &  =4  5: IF ?2 NE -1 6  PR  &  +D  %ZSUM
=ZSUM+KRELOP(Z%  IND-1  %,Z%  IND  %,%  LM  %,%  FX  %)%  &  =10  6: +D  %CALL L
EFBND(%  FX  %,%  ?2  %,Z%  IND  %,YBOUND,ZBONDS,YSTORE,ZSTORE)%  &  10: PC  +D
%IF(Z%  IND-1  %.EQ.ZSUM)Z%  IND  %=1%  &  =4  1: IF ?2 LT 0 7  IF ?3 NE 0 8  P
R  &  ZO  %IF(Y%  IND-1  RL  FN2  %)Z%  IND  %=1%  &  =4  8: IF ?3 NE -1 9  PR
&  +D  %ZSUM=ZSUM+KRELOP(Z%  IND-1  %,Z%  IND  %,%  LN  %,%  FX  %)%  &  =10  9:
+D  %CALL LEFBND(%  FX  %,%  ?3  %,Z%  IND  %,YBOUND,ZBONDS,YSTORE,ZSTORE)%  &
PR  &  +D  %ZSUM=ZSUM+MRELOP(Y%  IND-1  %,Z%  IND  %,%  LN  %)%  &  =10  7: PL
&  PR  &  +I  %IF(MARKER.NE.0) GOTO &  £  &  ZO  %IF(Y%  IND-2  RL  %Y%  IND-1
%)Z%  IND  %=1%  &  +I  %GOTO %  £+5  &  £-1  %ZSUM=ZSUM+NRELOP(Y%  IND-2  %,Y%
IND-1  %,%  LN  %)%  &  PC  ZO  %IF(ZSUM.LO.Z%  IND-1  %)Z%  IND  %=1%  &  £  %C
```

```
ONTINUE%  &  =4


   MEMBER    (NO.12)

  %CALL MEMBAR(YROWL,YROWR,ZTEMP,ZY,ZYPTR)%  &  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYP
TR-1%  &  DS  %ZTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #

   ENCODE    (NO.13)

  PR  &  %CALL NCOAD(Y%  IND  %,YROWL,ZY,ZYPTR,ZCBNDS,ZCPTR,ZTEMP)%  &  ZY  %ZY(
ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #

   DECODE    (NO.14)

  +D  %CALL DCODE(Z%  IND  %,YROWL,YROWR,ZCPTR)%  &  %Z%  IND  S  #


   NAND,NOR    (NOS.17-18)

  PL  &  PR  &  +D  %Z%  IND  %=0%  &  %IF(Z%  IND-1  %.NE.1%  AO  %Z%  IND-2  %
.NE.1)Z%  IND  %=1%  &  %Z%  IND  S  #

   SETTING OF ZCOORD STACK   (NO.19)

  %IF(ZCDPTR.GT.ZLIM14) CALL GVOVER(14,&%  +I  £  %)%  &  %ZCOORD(ZCOPTR)=%  R1
 &  £  %CONTINUE%  &  #

   RESETTING DF ZCOORD STACK   (NO.20)

  %ZCOORD(ZCDPTR)=0%  &  %ZCDPTR=ZCDPTR-1%  &  #

   +,-,*,/,**   (NOS.21-25)

  OR  LO  O  RO  CB  S  #

   MAXIMUM   (NO.26)

  PL  %  PR  &  +D  %Y%  IND  %=Y%  IND-2  &  %IF(Y%  IND-1  %.GT.Y%  IND-2  %)Y
%  IND  %=Y%  IND-1  &  %Y%  IND  S  #

   MINIMUM   (NO.27)
  PL  & PR  & +D  %Y%  IND  %=Y%  IND-1  & %IF(Y%  IND-2  %.LT.Y%  IND-1  %)Y
%  IND  %=Y%  IND-2  &  %Y%  IND  S  #

  RESIDUE   (NO.28)

  PL  & PR  & +D  %Y%  IND  %=Y%  IND-1  &  %IF(Y%  IND-2  %.NE.0)Y%  IND  %=Y
%  IND  %-ABS(Y%  IND-2  %)*AINT(Y%  IND-1  %/ABS(Y%  IND-2  %))%  &  %Y%  IND
S  #

  OPERAND-B   (NO.29)

  IF ?1 LT 0 1  IF ?2 NE 0 2  +D  %CALL OP1(%  ZM  %,%  FN1  %,Z%  IND  %,YROWL,
YROWR,ZCBNDS,ZCPTR,ZBONDS)%  &  IF Z NE 2 #  %Z%  IND  S  #  2: IF Z GT 1 3  IF
?2 NE -5 4  %CALL OPZL3(%  FX  %,%  ?3  %,YROWL)%  &  =6  4: IF ?2 NE -1 5  +D
%ZTYPE=0%  &  %CALL LITEST(%  FX  %,Z%  IND  %)%  &  %ZTYPEL(4)=NAMES(Z%  IND  %
+1)%  &  %CALL OPZL4(Z%  IND  %,ZTYPE,YROWL)%  &  =6  5: %CALL OPZL2(%  FX  %,ZC
BNDS,ZCPTR,ZBONDS,ZSUB,ZDIM,YROWL,YSTORE,ZSTORE)%  &  =6  3: IF ?2 NE -5 7  %CAL
L OPZR3(%  FX  %,%  ?3  %,YROWR)%  &  6: +D  %CALL BNDS(%  FX  %,%  ?2  %,ZROWDM
,ZCBNDS,ZCPTR,ZBONDS,ZTYPE,%  ZM  %)%  &  IF Z NE 2 #  ZZ  ZO  £  Z+  +D  %Z%  I
ND  %=Z%  IND-2  %+Z%  IND-1  &  %IF(Z%  IND  %.GT.ZLIM1) CALL GVOVER(1,&%  +I
£  %)%  &  %ZTEMP(Z%  IND  %)=INDX(YROWL,YROWR,Z%  IND-1  %)%  &  %IF(Z%  IND-1
 %.LT.ZTYPE) GOTO %  £-1  &  %ZYPTR=ZYPTR+1%  &  %IF(ZYPTR.GT.ZLIM7) CALL GVOVER
(7,&%  £  %)%  &  %ZY(ZYPTR)=Z%  ZC  %+Z%  IND-1  &  £  %CONTINUE%  &  ZY  %ZY(Z
YPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #  7: IF ?
2 NE -1 10  %ZTYPE=0%  &  %CALL LITEST(%  FX  +D  %,Z%  IND  %)%  &  %ZTYPEL(4)=
NAMES(Z%  IND  %+1)%  &  %CALL OPZR4(%  FX  %,ZTYPE,YROWR)%  &  =6  10: %CALL OP
ZR2(%  FX  %,ZCBNDS,ZCPTR,ZBONDS,ZSUB,ZDIM,YROWR,YSTORE,ZSTORE)%  &  =6  1: PR
&  %CALL OP3(%  ZM  %,Y%  IND  %,Z%  ZI  %,YROWR)%  &  PC  %CALL CBSET(%  ZM  %,
Z%  IND  %,ZCBNDS,ZCPTR,YROWL,YROWR,ZTEMP,ZY,ZYPTR)%  &  IF Z NE 2 #  +I  %IF(MA
RKER.NE.0) GOTO %  £  &  %ZA=Z%  IND-1  &  +I  %GOTO %  £  &  £-1  +I  ZY  %ZY(Z
YPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZA=ZTEMP(Z%  ZI  %+Z%  ZC  %)%  &  £-2  %C
ONTINUE%  &  %ZA%  S  #


  +D  %CALL YRHO(Z%  IND  %,ZROW)%  &  IF ?1 LT 0 1  IF ?2 NE 0 2  +D  %Y%  IND
 %=%  FN1  &  %CALL RHO2(Y%  IND  %,Z%  IND-1  %,ZROW,YTEMP,ZYY,ZYYPTR,ZCBNCS,ZC
PTR)%  &  5: ZS  %ZYY(ZYYPTR)=0%  &  %ZYYPTR=ZYYPTR-1%  &  DS  50: %YTEMP(Z%  ZI
 %+Z%  ZC  %)%  S  #  2: IF ?2 NE -1 4  %CALL RHO5(%  FX  %,Z%  IND  %,ZY,ZYPTR
,ZROW,ZCBNDS,ZCPTR,ZTEMP)%  &  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS2 %ZT
EMP(Z%  ZI  %+Z%  ZC  %)%  S  #  4: IF ?2 NE -3 6  %MARKER=-3%  &  11: %CALL RHO
4(ZROW,ZCBNDS,ZCPTR)%  &  =50  6: IF ?2 EQ -5 9  %CALL VECASE(%  FX  %,ZYY,ZYYPT
R,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,YTEMP,YSTORE)%  &  %CALL RHO4(ZROW,ZCBNDS,ZCPTR)%
&  =5  9: %CALL RHO1(%  FX  %,Z%  IND  %,ZYY,ZYYPTR,YTEMP,ZCBNDS,ZROW)%  &  =5
```

```
1: PR+  &  +I  %IF(MARKER.NE.0) GOTO %  £  &  %CALL RHO2(Y%  IND  %,Z%  IND-1  %
,ZROW,YTEMP,ZYY,ZYYPTR,ZCBNDS,ZCPTR)%  &  +I  %GOTO %  £  &  £-1  +D  Z+  %IF(Z%
  IND  %.GT.ZLIM25) CALL GVOVER(25,&%  +1  £+5  %)%  &  %YTEMP2(Z%  IND  %)=Y%
IND-1  &  PC  £  %CALL RHO3(Z%  IND  %,Z%  IND-3  %,YTEMPZ,ZCBNDS,ZCPTR,ZROW,YTE
MP,ZYY,ZYYPTR)%  &  £-1  =5

   BINOMIAL COEFFICIENT   (NO.32)

  PL  &  PR  &  +D  %Y%  IND  %=GAMMA(Y%  IND-1  %+1)/GAMMA(Y%  IND-2  %+1) *GAM
MA(Y%  IND-1  %-Y%  IND-2  %)%  &  %Y%  IND  S  #

   CIRCULAR   (NO.34)

  PL  &  PR  &  %IF(Y%  IND-1  %.GT.0) GOTO %  £  &  +D  %Y%  IND  %=RINGN(Y%  2
: IND-2  %,Y%  IND-1  %)%  &  %GOTO %  £+1  &  £  +D  %Y%  IND  %=RINGP(Y%  IND-
2  %,Y%  IND-1  %)%  &  +I  £  %CONTINUE%  &  %Y%  IND  S  #

   DEAL   (NO.35)

  PL  &  PR  &  %CALL QUERY(Z%  IND  %,Z%  IND-1  %,ZTEMP,ZY,ZYPTR,ZBOOL,ZCBNDS,
ZCPTR)%  &  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZTEMP(Z%  ZI  %+Z%  Z
C  %)%  S  #


   NATURAL LOGARITHM   (NO.37)

  PL  &  PR  &  +D  %Y%  IND  %=ALOG(Y%  IND-1  %)/ALOG(Y%  IND-2  %)%  &  %Y%
IND  S  #

   MONADIC +   (NO.39)

  %+%  RO  S  #

   NEGATIVE   (NO.40)

  +D  %Y%  IND  %=-%  RO  &  %Y%  IND  S  #

   SIGNUM   (NO.41)

  PR  &  %IF(Y%  IND  %.NE.0) GOTO %  £  &  +D  %Y%  IND  %=0%  &  %GOTO %  £+1
 &  £  %Y%  IND  %=SIGN(1.0,Y%  IND-1  %)%  &  +I  £  %CONTINUE %  &  %Y%  IND
S  +I  #

   RECIPROCAL   (NO.42)
 %1/%  RO  S  #

  EXPONENTIAL    (NO.43)

 PR  &  %EXP(Y%  IND  %)%  S  #

  CEILING    (NO.44)

 +I  PR  &  %IF(Y%  IND  %.LT.0) GOTO %  £  &  %Y%  IND  %=Y%  IND  %+(1-1E-8)%
 &  £  %Y%  IND  %=AINT(Y%  IND  %)%  &  +I  %Y%  IND  S  #

  FLOOR    (NO.45)

 +I  PR  &  %IF(Y%  IND  %.GE.0) GOTO %  £  &  %Y%  IND  %=ABS(Y%  IND  %-(1-1E
-8))%  &  £  %Y%  IND  %=AINT(Y%  IND  %)%  &  +I  %Y%  IND  S  #

  MAGNITUDE    (NO.46)

 PR  &  %ABS(Y%  IND  %)%  S  #

  INDEX GENERATOR    (NO.47)

 IF ?2 NE 0 3  IF ?1 NE -10 1  3: PR  &  =2  1: +D  %Y%  IND  %=%  FN1  &  2: %
IF(Y%  IND  %.NE.0) GOTO %  +I  £  &  %MARKER=-3%  &  %ZCPTR=0%  &  %GOTO %  £+1
 &  £  %CALL IOTB(Y%  IND  %,ZTEMP,ZY,ZYPTR,ZCBNDS,ZCPTR)%  &  ZY  %ZY(ZYPTR)=0
%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #


  SIZE    (NO.49)

 IF ?1 GT 0 1  PR  &  +I  %IF(MARKER.NE.0)GOTO %  £  &  +I  %MARKER=-3%  &  %ZC
PTR=0%  &  %GOTO %  £+6  &  £-1  +I  %CALL MRHO(ZCBNDS,ZCPTR,ZCOORD,ZCDPTR,ZTEMP
,ZY,ZYPTR)%  &  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS2  %ZTEMP(Z%  ZI  %+
Z%  ZC  %)%  S  #  1: %CALL NRHO(%  FX  %,%  ?2  %,ZY,ZYPTR,ZTEMP,ZBONDS,ZCBNDS,
ZCPTR,ZCOORD,ZCDPTR)%  &  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %ZTEMP(Z
%  ZI  %+Z%  ZC  %)%  S  #

  FACTORIAL    (NO.50)

 PR  %+1%  &  %GAMMA(Y%  IND  %)%  S  #

  PI TIMES    (NO.52)
```

```
BR  %3.14159*%  RO  CB  S  #

 ROLL   (NO.53)

PR  &  %QUERY1(Z%  IND  %)%  S  #

 LOGARITHM   (NO.55)

PR  &  %ALOG(Z%  IND' %)%  S  #

 GOTO   (NO.57)

 IF ?2 EQ -3 #  IF ?1 LT 0 1  IF ?2 NE 0 2  %Z1= %  FN1  &  10: %GOTO 1000%  &
#  2: IF ?2 NE -4 3  %GOTO %  L1  &  #  3: IF ?2 NE -5 4  %Z1=FUN(I%  FX  %)%
&  =10  4: V  %ZI=YSTORE(Z%  IND  %)%  &  =10  1: %IF(MARKER.EQ.-3) GOTO %  +I
£  &  %IF(MARKER.NE.0) GOTO %  +I  £  &  %Z1=%  RI+  &  %GOTO 1000%  &  £  %Z%
ZI  %=1%  &  %Z1=%  R1  &  %GOTO 1000%  &  £-1  %CONTINUE%  &  #

 NOT   (NO.58)

PR  &  Z0  %IF(Z%  IND-1  %.EQ.0) Z%  IND  %=1%  &  %Z%  IND  S  #

 I-BEAM   (NO.59)

PR  &  %IBEAM(Z%  IND  %)%  S  #

 GRADE-UP,GRADE-DOWN   (NOS.60-61)

 IF ?1 LT 0 10  IF ?2 NE 0 11  +D  %Y%  IND  %=%  FN1  &  %Y%  IND  S  %MARKER=
0%  &  #  11: +I  +D  £  +I  Z+  %IF(Z%  IND  %.GT.ZLIM18) CALL GVOVER(18,&%  £
%)%  &  %YGRAD(Z%  IND  %)=%  RO  &  IF ?2 NE 1 12  16: FL  15: £  %CONTINUE%
&  %CALL GRAD(%  IF 0 EQ 61 13  %0%  =14  13: %1%  14: %,YGRAD,ZTEMP,ZY,ZYPTR,ZC
8NOS,ZCPTR)%  &  IF MR EQ 1 17  ZY  %ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  DS  %Z
TEMP(Z%  ZI  %+Z%  ZC  %)%  S  #  12: IF ?2 NE -5 18  %IF(Z%  IND  %.LT.%  ?3  %
)) GOTO %  £  %  =15  18: %IF(Z%  IND  %.LT.LITNO(%  FX  %)) GOTO %  £  &  =15
10: +I  %IF(MARKER.EQ.0) GOTO %  £  &  +I  +D  £  +I  Z+  %IF(Z%  IND  %.GT.ZLIM
18) CALL GVOVER(18,&%  £+5  %)%  &  %YGRAD(Z%  IND  %+Z%  ZC  %)=%  R1  &  PC  M
R1  =15  17: %GOTO %  £  %  £-2  ZY  Z1  %ZTEMP(Z%  IND  %+Z%  ZC  %)=%  R1  &
%ZY(ZYPTR)=0%  &  %ZYPTR=ZYPTR-1%  &  MRO  £  DS  %ZTEMP(Z%  ZI  %+Z%  ZC  %)%
S  #

 OP.SQ.BRACKET,(   (NOS.62,64)

 %ZB%  INE  %=ZPOINT(ZPT)%  &  %ZPT=ZPT+1%  &  NB  +B  SJK  #
```

CL.SQ.BRACKET   (NO.63)

```
  %ZINDX(ZB%  INE  +N  %+%  N  %)=%  R1  &  %ZPOINT(ZPT)=ZB%  INE  %+%  N  &  SL
2  +D  IF N-1 NE 1 1  %CALL FIND1(%  FV  %,%  FX2  %,Z%  IND  %,Y%  IND  %,Z%  I
ND-2  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YBOUND,ZINDX,ZSTORE,ZCOORD,ZCDPTR,YST
ORE)%  &  %YSTORE(Z%  IND  %)%  S  =3  1: IF N-1 NE -5 2  %CALL FIND2(%  FV  %,%
  FX2  %,Z% IND  %,Y%  IND  %,Z%  IND-1  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YB
OUND,ZINDX,ZSTORE)%  &  %Y%  IND  S  =3  2: %CALL FIND3(% FV  %,% FX2  %,Z%  IND
  %,Z%  IND-1  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YBOUND,ZINDX,ZSTORE,ZCOORD,Z
CDPTR)%  &  %Z%  IND  S  3: IF MR EQ 1 4  %ZPT=ZPT-1%  &  #  4: %ZSAVE=1%  &  #
```

; IN INDEXING   (NO.66)

```
+N  %ZINDX(ZB%  INE  %+%  N  %)=%  R1  &  BP  RE  #
```

OUTER PRODUCT   (NO.67)

```
ZT  ZO  ZO  £  +I  Z+  ZO  £  +I  Z+  %Z%  IND-2  %=Z%  IND-2  %+1%  &  %YROWL(Z
%  IND-1  %)%  S  %YROWR(Z%  IND  %)%  S  RCM  #  %YTEMP(Z%  ZW-2  %+Z%  ZW-3  %
)=%  R1  &  %IF(Z%  ZW  %.LT.ZROWNA) GOTO %  £-1  &  %IF(Z%  ZW-1  %.LT.ZROWNO)
GOTO %  £-2  &  %ZYYPTR=ZYYPTR+1%  &  %IF(ZYYPTR.GT.ZLIM8) CALL GVOVER(8,&%  £+1
  %)%  &  +D  %CALL BOND(Z%  IND  %,ZCPTR,ZCBNDS)%  &  %ZYY(ZYYPTR)=%  ZW-5  %+Z
%  IND  &  ZS  %ZYY(ZYYPTR)=0%  &  %ZYYPTR=ZYYPTR-1%  &  DS  %YTEMP(Z%  ZI  %+Z%
  ZW-5  %)%  S  #
```

HETEROGENEOUS OUTPUT   (NOS.68,69,72)

```
  IF ?1 GT 0 1  %IF(MARKER.NE.0) GOTO %  +I  £  &  IF Y EQ 233 3  +D  %Y%  IND
%=%  R1+  &  %WRITE(6,101) Y%  IND  &  =10  3: +D  %Z%  IND  %=%  R1+  &  %WRITE
(6,103) Z%  IND  &  10: +I  %GOTO %  £+5  &  £-1  PR+  &  %CALL OUT1(Y%  IND  %,
ZCBNDS,ZCPTR)%  &  PC  £  %CONTINUE%  &  #  1: IF ?2 NE 0 6  IF X EQ 109 7  +D
%Y%  IND  %=%  FN1  &  %WRITE(6,101)Y%  IND  &  #  7: +D  %Z%  IND  %=%  FN1  &
 %WRITE(6,103)Z%  IND  &  #  6: %CALL OUT2(%  ?1  %,%  ?2  %,ZCBNDS,ZCPTR,ZBONDS
,ZSTORE,ZDIM,ZSUB,YSTORE)%  &  #
```

COMMENT   (NO.70)

```
LI  &  #
```

INNER PRODUCT   (NO.71)

```
 +I  ZT  %IF(ZCPTR.NE.O) GOTO %  £  &  DS  %YTEMP(Z%  IND  %+Z%  ZC  %)%  S  %Y
ROWL(Z%  IND  %)%  S  %YROWR(Z%  IND  %)%  S  RCM  #  RCM  #  %YTEMP(Z%  IND  %+
Z%  ZC  %)=%  R1  &  %IF(Z%  IND  %.LE.ZROWDM) GOTO %  £  &  %ZYYPTR=ZYYPTR+1%
&  %IF(ZYYPTR.GT.ZLIM8) CALL GVOVER(8,&%  +I  £  %)%  &  %ZYY(ZYYPTR)=Z%  ZC  %+
1%  &  %GOTO %  £  &  £-2  ZO  +I  £  Z+  %IF(Z%  IND  %.GT.ZLIM28) CALL GVOVER(
28,&%  £+15  %)%  &  %ZTEMP3(Z%  IND  %)=1%  &  %IF(Z%  IND  %.LT.ZCPTR) GOTO %
 £  &  %Z%  IND  %=0%  &  ZO  +D  %Z%  IND  %=ZCPTR-1%  &  +I  ZO  £  Z+  %Z%  I
ND-2  %=Z%  IND-2  %+1%  &  ZO  +I  £  Z+  +D  %Z%  IND  %=(Z%  IND-1  %-1)*ZCBN
DS(ZCPTR)+Z%  IND-2  &  %YTEMP(Z%  IND-4  %+Z%  ZC  %)%  S  %YROWL(Z%  IND-1  %+
Z%  IND-5  %)%  S  %YROWR(Z%  IND  %)%  S  RCM  #  RCM  #  %YTEMP(Z%  ZW-4  %+Z%
  ZC  %)=%  R1  &  %IF(Z%  ZW-1  %.LT.ZROWDM) GOTO %  £  &  %IF(Z%  IND-2  %.LT.
ZCBNDS(ZCPTR)) GOTO %  £-1  &  +D  %Z%  IND  %=Z%  IND-4  %+1%  &  +I  £  %ZTEMP
3(Z%  IND  %)=1%  &  Z+  %IF(Z%  IND  %.LE.ZCPTR) GOTO %  £  &  +I  £  %ZTEMP3(Z
%  IND-4  %)=ZTEMP3(Z%  IND-4  %)+1%  &  %IF(ZTEMP3(Z%  IND-4  %).LE.ZCBNDS(Z%
IND-4  %)) GOTO %  +I  £  &  +I  %IF(Z%  IND-4  %.EQ.1) GOTO %  £  &  %Z%  IND-4
  %=Z%  IND-4  %-1%  &  %GOTO %  £-2  &  £-1  %Z%  IND-4  %=ZCPTR-1%  &  %Z%  IN
D-6  %=Z%  IND-6  %+ZROWDM%  &  %Z%  IND-3  %=0%  &  %GOTO %  £-5  &  £  %CONTIN
UE%  &  %ZYYPTR=ZYYPTR+1%  &  %IF(ZYYPTR.GT.ZLIM8) CALL GVOVER(8,&%  £-7  %)%  &
  %ZYY(ZYYPTR)=Z%  IND-5  %+Z%  IND-8  &  ZS  %ZYY(ZYYPTR)=0%  &  %ZYYPTR=ZYYPTR
-1%  &  DS  £-8  %CONTINUE%  &  %YTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #

  :   (NO.73)

  T2  L1  T7  #

  OPENING DEL   (NO.74)

  IF I EQ 0 1  %FUNCTION %  IF N GT 3 2  FN2  %(%  IF N EQ 2 20  IF N3 NE 0 4  F
N3  5: %,%  20: %ZTEMP,YTEMP,YSTORE,YROWL,YROWR,ZCBNDS,ZCPTR,ZY,ZYPTR,ZYY,ZYYPTR
,ZBONDS,YBOUND,YBOND,ZPOINT,ZPT,ZINDX,ZBOOL,ZCOORD,ZCDPTR,ZSTORE,ZL,YGRAD,ZROW,Z
GRAD,ZDIM,ZSUB,YTEMP2,YTEM,XBOUND,ZTEMP3)%  &  CS  #  4: %ZF1%  =5  2: FN3  %(%
 IF N2 NE 0 6  FN2  %,%  IF N4 NE 0 7  FN4  =5  7: %ZF2%  =5  6: %ZF1,%  IF N4 N
E 0 7  FN4  =5  1: %SUBROUTINE %  IF N GT 2 9  FN1  %(%  IF N EQ 1 20  IF N2 NE
0 4  FN2  =5  9: FN2  %(%  IF N1 NE 0 10  FN1  %,%  IF N3 NE 0 7  FN3  =5  10: %
ZF1,%  IF N3 NE 0 7  FN3  =5

  CLOSING DEL   (NO.75)

  T2  %1000 IF(Z1.LE.0.OR.Z1.GT.%  MX  %) CONTINUE%  &  %GOTO (%  S1,MX-1  MX-S1
  %,%  S1  MX  %),Z1%  &  #
REDUCTION   (NO.76)

 IF ?1 LT 0 #  IF ?2 NE 0 2  FNI  S  %OPL=%  ID  &  %OPL%  S  RCM  #  #  2: IF
?2 EQ -3 3  IF ?2 NE 1 4  %CALL BOINFO(%  FX  %,ZDIMS,ZBOUND)%  &  %IF(ZDIMS.NE.
1) GOTO %  +I  £  &  %ZB%  INE  %=ZPOINT(ZPT)%  &  %ZPT=ZPT+1%  &  %ZINDX(ZB%  I
NE  %+1)=1%  &  %OPL=%  ID  &  %ZPOINT(ZPT)=ZB%  INE  %+1%  &  +I  £  +D  %Z%  I
ND  %=FIND(1%  FV2  FX  %,ZPOINT,ZPT,ZCBNDS,ZCPTR,YBOUND,ZINDX,ZSTORE,ZBONDS,ZCO
ORD,ZCDPTR)%  &  %OPR=YSTORE(Z%  IND  %)%  &  %OPR%  S  %OPL%  S  RCM  #  %OPL=%
 R1  &  %ZINDX(ZB%  INE  %+1)=ZINDX(ZB%  INE  %+1)+1%  &  %IF(ZINDX(ZB%  INE  %
+1).LE.ZBONDS(ZBOUND)) GOTO %  £  &  %MARKER=0%  &  ZT  Z1  %YTEMP(Z%  ZC  %+Z%
 IND  %)=OPL%  &  +I  %GOTO %  £  &  £-2  SL  %CALL BNDSET(1,%  FX  %,1,ZCBNDS,Z
CPTR,ZBONDS)             %  &  Z0  %IF(ZCOORD(ZCDPTR).EQ.0) ZCOORD(ZCDPTR)=ZDIMS%
 &  %ZCO=ZCOORD(ZCDPTR)%  &  ZT  %OPL=%  ID  &  +I  £  +D  %Z%  IND  %=FIND(1%
FV2  FX  %,ZPOINT,ZPT,ZCBNDS,ZCPTR,YBOUND,ZINDX,ZSTORE,ZBONDS,ZCOORD,ZCDPTR)%  &
  %OPR=YSTORE(Z%  IND  %)%  &  %OPR%  S  %OPL%  S  RCM  #  %OPL=%  R1  &  %ZINDX
(ZCO+ZB%  INE  %)=ZINDX(ZCO+ZB%  INE  %)+1%  &  %IF(ZINDX(ZCO+ZB%  INE  %).LE.ZB
ONDS(ZBOUND+ZCO-1)) GOTO %  £  &  %Z%  IND-2  %=Z%  IND-2  %+1%  &  %YTEMP(Z%  I
ND-2  %+Z%  IND-1  %)=OPL%  &  FL  %CALL REDBND(ZCO,ZCBNDS,ZCPTR)%  &  ZS  %ZYY(
ZYYPTR)=0%  &  %ZYYPTR=ZYYPTR-1%  &  DS  %YTEMP(Z%  ZC  %+Z%  Z1  %)%  S  #  3:
+D  %Z%  IND  %=%  ID  &  %Z%  IND  S  %MARKER=0%  &  %OPL=%  ID  &  %OPL%  S  R
CM  #  #  4: IF ?2 NE -5 #  SL  +D  %CALL FIND2(1%  FV2  FX  %,Z%  IND  %,Y%  IN
D  %,Z%  IND-1  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YBOUND,ZINDX,ZSTORE)%  &  %
OPL=%  ID  &  Z0  £  %OPR=Y%  IND-1  &  %OPR%  S  %OPL%  S  RCM  #  %OPL=%  R1
&  Z+  %IF(Z%  IND  %.LE.Z%  IND-2  %) GOTO %  £  &  %OPL%  S  %MARKER=0%  &  #

 LG  LS  IF MAX EQ 0 1  %RETURN%  &  #  1: L1,1  IF L1 EQ -1 2  %GOTO %  L2,1
&  2: #

 PRODUCES 'Z1=<EXPRESSION>' 'GOTO 1000'   (NO.78)

 +D  %Z1=%  FNI  &  %GOTO 1000%  &  #

 OPERAND-A   (NO.80)

 IF ?1 LT 0 #  IF F EQ 1 1  IF ?2 EQ -5 2  IF ?2 GT -2 2  3: FNI  S  #  2: IF ?
2 EQ 0 3  IF A NE 0 6  SL  £-1  6: +D  IF ?2 NE 1 4  %CALL FIND1(%  FV  %,%  FX
 %,Z%  IND  %,Y%  IND  %,Z%  IND-6  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YBOUND,
ZINDX,ZSTORE,ZCOORD,ZCDPTR,YSTORE)%  &  %Y%  IND  S  #  4: IF ?2 NE -5 5  %CALL
FIND2(%  FV  %,%  FX  %,Z%  IND  %,Y%  IND  %,Z%  IND-5  %,ZNC,ZCBNDS,ZCPTR,ZBON
DS,ZPOINT,ZPT,YBOUND,ZINDX,ZSTORE)%  &  %Y%  IND  S  #  5: %CALL FIND3(%  FV  %,
%  FX  %,Z%  IND  %,Z%  IND-5  %,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YBOUND,ZINDX
,ZSTORE,ZCOORD,ZCDPTR)%  &  %Z%  IND  S  #  1: FA  S  &ZTEMP,YTEMP,YSTORE,YROWL,
YROWR,ZCBNDS,ZCPTR,ZY,ZYPTR,ZYY,ZYYPTR,ZBONDS,YBOUND,YBOND,ZPOINT,ZPT,ZINDX,ZBCO
L,ZCOORD,ZCDPTR,ZSTORE,ZL,YGRAD,ZROW,ZGRAD,ZDIM,ZSUB,YTEMP2,YTEM,XBOUND,ZTEMP3)&
```

#

QUAD INPUT   (NO.81)

```
 +I  %READ 102,ZVBND%  &  %IF(ZVBND.NE.-1) GOTO %  £  &  %WRITE(6,106)%  &  %MA
RKER=-3%  &  +I  %GOTO %  £+3  &  £-1  %IF(ZVBND.NE.0) GOTO %  +I  £  &  %MARKER
=0%  &  %READ(5,104) YTEM(1)%  &  %WRITE(6,104) YTEM(1)%  &  %GOTO %  £+2  &  £
 %IF(ZVBND.GT.ZLIM26) CALL GVOVER(26,&%  £+2  %)%  &  +I  +D  %READ(5,104) (YTEM
(Z%  IND  %),Z%  IND  %=1,ZVBND)%  &  %WRITE(6,104) (YTEM(Z%  IND  %),Z%  IND  %
=1,ZVBND)%  &  %ZCPTR=1%  &  %ZCBNDS(1)=ZVBND%  &  DS  %YTEM(Z%  ZI  %)%  S  #
```

QUOTE-QUAD INPUT   (NO.82)

```
 +I  %READ 102,ZVBND%  &  %IF(ZVBND.NE.-1) GOTO %  £  &  %WRITE(6,106)%  &  %MA
RKER=-3%  &  +I  %GOTO %  £+3  &  £-1  %IF(ZVBND.NE.0) GOTO %  +I  £  &  %MARKER
=0%  &  %READ 105,ZL(1)%  &  %WRITE(6,105) ZL(1)%  &  %GOTO %  £+2  &  £  %IF(ZV
BND.GT.ZLIM22) CALL GVOVER(22,&%  £+3  %)%  &  +D  %READ(5,105) (ZL(Z%  IND  %),
Z%  IND  %=1,ZVBND)%  &  %WRITE(6,105) (ZL(Z%  IND  %),Z%  IND  %=1,ZVBND)%  &
%ZCPTR=1%  &  %ZCBNDS(1)=ZVBND%  &  DS  %ZL(Z%  ZI  %)%  S  #
```

HANDLES LEFT OPERAND OF DYADIC RHO   (NO.83)

```
 IF ?1 LT 0 1  IF ?2 NE 0 2  %ZROW(1)=%  FN1  &  %ZROWNO=1%  &  #  2: %CALL RHO
BND(%  FX  %,%  ?2  %,ZROW,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,YSTORE,ZDIM,ZSUB)%  &  #
1: +I  %IF(MARKER.NE.0) GOTO %  £  &  %ZROW(1)=%  RI+  &  %ZROWNO=1%  &  +I  %GO
TO %  £  &  £-1  PR  &  %IF(Z%  ZI  %.GT.ZLIM20) CALL GVOVER(20,&%  £  %)%  &  %
ZROW(Z%  ZI  %)=Y%  IND  &  PC  %ZROWNO=Z%  IND  &  £-1  %CONTINUE%  &  #
```

UPDATES IFIND STACK   (NO.84)

```
FP  #
```

UPDATES ZCOORD STACK POINTER   (NO.85)

```
%ZCOPTR=ZCOPTR+1%  &  #
```

UNSTACKS IFIND STACK VALUE   (NO.86)

```
FS  #
```

```
  SETS ZCOORD STACK VALUE TO 1   (NO.87)

 %ZCOORD(ZCOPTR)=1%  &  #

  PRODUCES 'CALL ' BEFORE SUBROUTINE NAME   (NO.88)

 %CALL %  #

  PRODUCES CALL OF SCFIND   (NO.89)

 +I  IF O NE 31 1  2: IF A NE 0 3  SL  =3  1: IF O EQ 49 2  IF O EQ 2 2  IF O E
Q 20 2  3: +D  £-1  %Y%  IND  %=SCFIND(%  FV  %,%  FNI  %,ZPOINT,ZPT,ZCBNDS,ZCPT
R,YBOUND,ZINDX,ZSTORE,ZBONDS)%  &  %Y%  IND  S  #

  PRODUCES NON-EXECUTABLE STATEMENTS   (NO.90)

 CS  #

  CLOSING SQUARE BRACKET (LOOPS)   (NO.91)

 ZZ  +D  %CALL BDVAL(%  ?(-1)  %,%  BB  %,Z%  IND  %)%  &  +I  £  %Z%  ZE  %=Z%
 ZE  %+1%  &  %ZTEMP(Z%  ZC  %+Z%  ZE  %)=%  RI  &  %IF(Z%  ZE  %.LT.Z%  IND  %
) GOTO %  £  &  %ZYPTR=ZYPTR+1%  &  %ZY(ZYPTR)=Z%  ZE  &  ZY  %ZSAVE=0%  &  DS
%ZTEMP(Z%  ZI  %+Z%  ZC  %)%  S  #

  ; (LOOPS)   (NO.92)

 BB  +D  %Z%  IND  %=Z%  IND-1  %+%  +N  N  &  ZA  %ZINDX(Z8%  INE  %+%  N  %)=
Z%  IND  &  #

  FORMATS   (NO.16)

 T3  %101 FORMAT(1X,G12.6)%  &  T3  %102 FORMAT(G12)%  &  T3  %103 FORMAT(1X,I1
2)%  &  T3  %104 FORMAT(10G12.6)%  &  T3  %105 FORMAT(80A1)%  &  T3  %106 FORMAT
(1X,/)%  &  #

  CLOSING DEL 2   (NO.31)

 %CALL LOGREM%  &  IF I EQ 0 1  FO  T3  %100 FORMAT(1X,G12.6)%  &  1: %RETURN%
&  %END%  &  #

  CLOSING DEL 3   (NO.33)
  T2  %1000 IF(Z1.LE.0.OR.Z1.GT.%  MX+1  %) CONTINUE%  &  %GOTO(%  S1,MX  MX-S1+
1  %,%  &1  MX+1  %),Z1%  &  #

   PRODUCES CODE   Z<INTEGER>=0   (NO.36)

  ZD  #

   STORES SUBSCRIPT CODE   (NO.38)

 ZZ  %Z%  ZE  %=Z%  ZE  %+1%  &  %ZTEMP(Z%  ZC  %+Z%  ZE  %)=%  R1  &  PC  %ZYP
TR=ZYPTR+1%  &  %ZY(ZYPTR)=Z%  ZE  &  ZY  %ZSAVE=0%  &  DS  %ZTEMP(Z%  ZI  %+Z%
 ZC  %)%  S  #
   ; LOOPS

  %Z%  ZE  S  #
```

APPENDIX 5

COMPARISON OF BRACKETING AND REVERSE POLISH METHODS

Three possible methods are considered:

Method 1 involving two distinct processes :

(a) a lexical scan

(b) a right-to-left scan involving both production of reverse polish and expansion of macros to generate the target-language code.

Method 2 involving three distinct processes :

(a) a lexical scan

(b) a right-to-left scan in which reverse polish is produced.

(c) a left-to-right scan in which the target language code is generated.

Method 3 involving three distinct processes :

(a) a lexical scan

(b) a right-to-left scan in which a bracketed intermediate code form is produced

(c) a left-to-right scan in which the target-language code is generated.

A description of the handling of "FIND call" operators is given in the main text. Using the method described, it is possible to obtain an increase in efficiency of the target-language code as compared with the original APL code. Method 3 is most suitable for the handling of these operators, for the reasons given below. (In fact, a great deal of work had been done on Method 3 before Methods 1 and 2 were considered. For expressions not involving "FIND call" operators, reverse polish methods would probably be slightly more efficient. However, the ease of handling of "FIND call" operators justifies the use of Method 3.)

Consider Method 1 applied to expressions such as that given below.

$$F \leftarrow A/B + (C-D*E \uparrow D) - C$$

For both / and ↑, the index and type value for the left operand (or the value and type value, if the type value is ∅) is required. If this information is extracted during the lexical scan, then the lexical scan for this method would necessarily be more complex than the lexical scan for Method 3, as more tests would be required.

For example, the expression involves two "FIND call" operators. The above information is required for each, together with the "scope" of each operator, that is, the extent of influence for each operator.

The required value for the first parameter of the "FIND call" must also be retained here.

All the necessary information could be retained by replacing the 1-byte entries for operators by 3-byte entries, giving

1. the negative of the operator macro number
2. the value for the first parameter of the FIND call
3. the index value for the left operand (to be inserted in the FIND call).

Some means would have to be devised of distinguishing these entries from the 2-byte operand entries.

It is better not to store the type value for the left operand of the FIND call at this stage, as type values can vary dynamically.

It is difficult to keep a record of the scope of an operator using Method 1. In the above example, the scope of ↑ is terminated by) . However, in the following example,

$$A/(B + (C*D) + (F \uparrow D) + G) - E\ ,$$

the scope of ↑ is terminated by the second), but the scope of / is not terminated until the end of the line is reached. This implies that a bracket count is necessary for the handling of "FIND call" operators, but that tests must also be made to detect the occurrence of operators whose scope extends to the end of the line.

Thus, it can be seen that the lexical scan for this method would be more complex than that described for Method 3 in the main text.

Method 1 has the advantage that one scan can be eliminated. However, the lexical scan and the right-to-left scan would both be more complex than for Methods 2 and 3.

Now consider Method 2. Using this method, the first parameter value and the left operand (for "FIND call" operators) could be retained during the right-to-left scan, in which the reverse polish notation is generated.

When a "FIND call" operator is detected, it is known that all preceding operators (either from the right-most end or from a previous occurrence of a "FIND call" operator, with due regard for bracketing) lie within the scope of the operator. For example, in the statement

F ← A/B + (C-D*E ↑ D+B) ÷ C ,

the operators + (first occurrence), −, *, ÷, lie within the scope of /, while the operator + (second occurrence) lies within the scope of ↑ . The operator ← lies outwith the scopes of both / and ↑ .

The above information must be stored in some form. Thus, the complexity of the right-to-left scan would not be considerably less for Method 2 than for Method 3.

The handling of "FIND call" operators is more difficult using Method 2, because brackets, which are important to the method, are discarded during the right-to-left scan. For example, consider the APL expression

A/(B ÷ (C ↑ D)) − E .

If a reverse polish notation is produced during the right-to-left scan, the operator / is not reached until (B ÷ (C ↑ D)) − E has been handled. At this stage, the brackets have been discarded, and thus the scope of / cannot easily be determined.

There is little difference in complexity between the left-to-right scans of Methods 2 and 3.

Method 3 has been discussed in detail in the main text. There is no problem with "FIND call" operators using this method as the necessary information can be obtained easily during the left-to-right scan. Using Method 2, it is not possible to delay the storage of the required information until the left-to-right scan, as the brackets present have already been discarded. Thus, due to the problem posed by the need for a bracket count, if a reverse polish method is to be used, it would appear that Method 1 is preferable.

Comparison of Complexity

Lexical Scan	Method 3	≑	Method 2	<	Method 1
Right-to-Left Scan	Method 3	≑	Method 2	<<	Method 1
Left-to-Right Scan	Method 3	≑	Method 2		None for Method 1

APPENDIX 6

A list of restrictions on the types of APL statement able to be translated is now given. The list is in two parts:

A. Important restrictions

B. Less important restrictions.

List A is further sub-divided into

(i) those restrictions imposed as a result of the method of conversion

(ii) those restrictions which could be removed using the same conversion method.

A. IMPORTANT RESTRICTIONS

(i)

1. Function or subroutine parameters must be either

   (a) numeric scalars

   or (b) numeric non-scalar variable names

   or (c) literal variable names

The above parameter types are the only ones possible in a function or subroutine definition header statement. The type assigned to parameters in the header statement determines the code to be generated in the function or subroutine body. Thus, for example, a literal constant cannot be used as an actual parameter when a literal variable name has been used in the header statement. Similarly, if B is a numeric non-scalar variable name, then the following code would not be handled correctly,

```
∇  A   FN   B
   ___________
   ___________
   ___________
       ∇
   ___________
   ___________
 X   FN   3
   ___________
   ___________
```

In the above example, looping code would be generated for non-scalar accessing and this would be incorrect if the variable B was replaced by the constant 3.

2. Use of non-scalar or literal parameters requires a knowledge of the storage method used by the conversion routines. For example, consider the function

```
∇ R ← CTD   L
      ⋮
      ∇
```

where L is a literal variable name.

To call CTD with parameter 'ABCD', it is necessary to

(a) first assign 'ABCD' to a literal variable name, say L1.

(b) set up an entry in NAMES for 'ABCD'.

(c) set up an entry in NAMES for L1.

(d) set up an entry in LITBLE associating 'ABCD' with L1.

(e) call CTD with parameter value equal to the NAMES index for L1.

3. Function result variables must be numeric scalars. This is due to the fact that the result value has to be assigned to the function name.

4. Recursive function calls have not been catered for.

5. The left parameter of a "FIND call" operator cannot be an expression. This is due to the method of handling certain mixed functions. The problem can be avoided by introducing an extra variable name. For example, (A+B) ⌽ X cannot be handled, but

```
R ← A+B
R ⌽ X
```

will be translated correctly.

6. Expressions such as

```
A[Y] ← Y ← B+C
```

will not be translated correctly, as immediate action macros are expanded to handle indexing. Thus, if Y originally has value 4, then A[4] (not A[B+C]) will be altered. This problem can be avoided by splitting the expression into two parts, that is,

```
Y ← B+C
A[Y] ← Y
```

(Multiple assignments will be handled correctly as long as the left operand of an assignment is not used in an indexed expression on the same line.)

(ii)

1. At present, no account has been taken of run-time changes in the type values associated with variables. The type values will be updated as required. However, the conversion routines use instructions of the form

```
IF   ?n  <rel-op.>   m   k
```

to test the type values.

In fact, the macro bodies should be altered to test the appropriate NAMES entries at run-time.

2. Nesting of non-scalar indices has not been handled. Thus, for example, A[3 4 5] is acceptable, but A[3; B[C ; D ; E]] where C or D or E is non-scalar, is not allowed. This restriction has been imposed merely to avoid production of unwieldy code.

3. The operand for the reduction operator cannot be an expression. This restriction may be removed by updating the reduction macro (see Appendix 4).

4. A co-ordinate value may not be specified for the following functions

   (a) dyadic rho

   (b) monadic comma

   (c) dyadic iota

   (d) grade-up

   (e) grade-down

   (f) member

5. The variable name MARKER is reserved and should not be used in the APL source. This restriction may be removed by replacing the name MARKER by, for example, ZMARK in the macro bodies and all the run-time routines contained in SARUN.

6. Non-scalar variable names may be used as parameters for ←, but non-scalar expressions are not allowed.

7. Locked functions are not handled.

8. 1-indexing is assumed throughout.

9. The functions

```
-5 ○ B

-6 ○ B

-7 ○ B
```

have not been defined.

An exponential series is required to handle these functions. The user may supply the appropriate series to any required degree of accuracy. This involves updating the function RINGN, which is present in module library SARUN. If no series is provided by the user, a zero value will be returned. A message is also printed out for the user.

B. LESS IMPORTANT RESTRICTIONS

This list is also divided into two parts. The first list contains restrictions which apply because the facilities to which they refer are system-dependent.

1. No system commands are dealt with.

2. No workspace size is defined.

3. No function editing facilities are available.

4. No trace or stop control is allowed.

5. Any constant will be represented in the output code exactly as it appeared in the input stream. Size restrictions will be imposed at run-time by the system used to run the converted routine.

6. I-Beam functions are replaced in the output stream by

   IBEAM (K) ,

   where K determines the function. The body of the function must be written by the user, since I-Beam functions are system-dependent.

The following size limitations also apply:

1. A function or subroutine may contain up to 99 lines of code.

2. In a set of supplied functions or subroutines, there may be up to 64 non-scalar variable names.

3. The array NAMES has 5ØØØ locations. Therefore, there is a limit to the number of entries which may be placed there. Garbage collection of NAMES (by calling subroutine NGARB) may ease the situation .

4. The intermediate code form may occupy up to 2ØØ bytes. At present, 6 spaces are left for insertion of brackets when certain symbols are recognised. This amount may be varied. (See Chapter III.)

5. A macro body may contain up to 1ØØ labelled statements.

6. A macro body may occupy up to 4ØØØ bytes.

7. The label table, LTABLE, may hold up to 1ØØ entries. A set of routines should therefore contain no more than 1ØØ labels.

8. A constant vector (with one blank separating each element and one terminating blank) may contain up to 3ØØ characters in all.

9. There may be up to 1Ø "locked" local variable names at any stage. (See Chapter II.)

10. There may be up to 1ØØ long names (that is, identifier names having more than 6 characters) in any set of routines.

11. Nesting of brackets is allowed up to a maximum of 4Ø levels deep in APL expressions.

12. Nesting of indexed expressions is allowed up to a maximum of 5 levels deep.

13. The parameter stack, IDSTK, may contain up to 5ØØ entries.

14. 1Ø locations are set aside for storage of result variable and function name. This allows up to 5 functions per set of routines, but does not restrict the number of subroutines.

15. there may be up to 1Ø local variables in a set of routines.

16. A maximum of 1Ø routines may be converted at once, assuming condition 14 is satisfied.

17. Nesting of "FIND call" operations is allowed up to a maximum of 1Ø.

18. Nesting of bracketed expressions is allowed in macro bodies up to a maximum of 1Ø levels deep.

APPENDIX 7

EASE OF CONVERSION TO OTHER LANGUAGES

The generated code was produced in FORTRAN, since that language was most often available for testing purposes. Had ALGOL or PL/1, for example, been as readily available, they might equally as well have been used.

Code is generated in the following ways:

1. It is generated as a result of macro expansions.
2. There is a library (SARUN) containing the object modules of subroutines to be included during execution of the converted routines.

All the subroutines and functions contained in SARUN were written in FORTRAN. However, since the object modules are included during execution of the converted routines, it is not necessary to convert these to other languages.

Thus it is only necessary to consider conversion to other languages of code generated as a result of macro expansions. In particular, the languages ALGOL and PL/1 are considered. As far as possible, only the types of FORTRAN statements having counterparts in ALGOL and PL/1 were generated using macro expansions. Thus the problem of conversion to either of these two languages instead of FORTRAN is fairly straightforward.

The generated code takes two forms:

(a) explicitly generated code, using the macro instructions

%---- CODE ----% &

(see Chapter V).

(b) implicitly generated code. This is produced using composite macro instructions, such as SL, FL, CS. All the APL-FORTRAN conversion routines are contained in the module library SALIB. To convert implicitly generated code to other languages, the following subroutines in SALIB require to be updated.

AREC
CSREC
DIMC
DOLLAR
DREC
FAVECT
FLOOPS
FSTORE
MKSET
NSCALE
FREC
FCODE
PREC
RLREC
SLOOP
SREC
STPSET
VREC
ZREC

The subroutines listed above (with the exceptions AREC, RLREC and DOLLAR) all contain LOGICAL * 1 arrays in which the code to be generated is stored, character by character. Usually, a DO-loop is executed to place successive characters in the array MTEMP. Complete lines of code are produced in this way, and then transferred to the output medium.

Thus, to alter implicitly generated code, the LOGICAL * 1 arrays must be updated, and often also the associated DO-loops.

The subroutine AREC produces

.AND. or .OR.

while RLREC produces one of the forms

.EQ. , .LE. , .LT. , .GT. , .GE. , .NE.

These subroutines must also be updated if the target language is not FORTRAN.

DOLLAR is used in production of label numbers, (see 4.).

All other subroutines in SALIB are independent of the target-language and may be regarded as fixed. Thus the APL-FORTRAN conversion involves

1. fixed subroutines and functions in SALIB

2. the subroutines listed in (b) above

3. macro bodies, contained in the data-set SAMACBOD.

To convert APL routines to ALGOL or to PL/1, it is necessary to combine 1. above with updated versions of 2. and 3.

It is possible to produce entire libraries for APL-ALGOL or APL-PL/1 conversion by making changes of the types listed below. It is then a simple matter to provide a user option by which the target-language is chosen by the user.

The types of changes to be made take the following forms:

1. Non-executable statements

1.1 IMPLICIT REAL (A-Y)
IMPLICIT INTEGER (Z-Z)

These statements were included for the ease they afforded in the introduction of non-ambiguous variable names.

(a) Conversion to ALGOL

There is no equivalent ALGOL statement type. In ALGOL, all variable names must be declared. This involves accumulating a list of all the variable names used in a routine and inserting the complete list at a later stage. All non-scalar variables are declared in FORTRAN, and their conversion may be handled as indicated in 1.2 .

All scalar global variables should be explicitly defined, together with all scalar variables in the original APL routine. In addition, all scalar variables of the forms

$$Z < i > \quad , \quad i = 1,---,N$$

and $$ZB < j > \quad , \quad j = 1,---,M$$

should be explicitly declared. At the end of the routines, the values N and M can be obtained from the variables IND and INE respectively.

(b) Conversion to PL/1

A similar process to that given in (a) above must be carried out.

1.2 INTEGER - - -

REAL - - -

LOGICAL*1 - - -

The equivalent ALGOL or PL/1 forms should be used instead. (It is not necessary to declare all variable names in PL/1, but it is better to do so to avoid unnecessary complications.)

1.3 COMMON - - -

These statements need not appear at all in the FORTRAN version if the variables concerned are placed in the parameter lists of the subroutines involved. This is referred to at the start of Chapter VIII.

(a) Conversion to ALGOL or PL/1

If COMMON statements are present, the variable names concerned should be placed in the appropriate parameter lists.

1.4 EQUIVALENCE - - -

These statements were used to facilitate character handling in FORTRAN. They are not directly equivalent to any ALGOL or PL/1 statement types, but this does not matter as their use can be dispensed with in ALGOL or PL/1 . (Character handling is easier using these languages.) For example, instead of

```
        LOGICAL * 1  N(4),NAMES (5ØØØ)
        EQUIVALENCE (NI,N)
        NI  =  193
        NAMES (I)  =  N(4)
```

code of the form shown below (illustrated for ALGOL) may be used

```
        'STRING'   NAMES  (5ØØØ) ;
                   NAMES [I] : = 'A' ;
```

In ALGOL, the main program (which is partially produced during the conversion of a routine to FORTRAN) must be delimited by 'BEGIN' and 'END'. The equivalent structure in PL/1 is

```
        <label> : PROCEDURE OPTIONS (MAIN) ;
                      and
                  END  <label> ;
```

All changes of the types listed above may be made by altering the subroutines CSREC and DIMC (contained in SALIB).

2. Subroutine and function definitions

(a) Conversion to ALGOL

The form   SUBROUTINE - - - -

produced in the FORTRAN version must be altered to the form

'PROCEDURE' - - - - ;

while the form

FUNCTION - - - -

must be altered to the form

< type > 'PROCEDURE' - - - - ;

The form RETURN END after a FORTRAN subroutine or function should be replaced by 'END' ; .

(b) Conversion to PL/1

The form SUBROUTINE - - - should be replaced by

< label > : PROCEDURE OPTIONS (---) ;

while the form FUNCTION - - - should be replaced by

< label > : < type > PROCEDURE OPTIONS (---) ;

The statements RETURN END in FORTRAN should be replaced by END < label >; .

The above changes may be made by updating macro bodies 74 and 31 (contained in the data set SAMACBOD).

3. READ - - -

WRITE - - -

FORMAT - - -

All I/O statements should be converted to the forms in use at the particular ALGOL or PL/1 installations where the converted routines are to be run.

These alterations may be made by updating macro numbers 68, 81, 82, 16 and 31.

4. Label numbers must be replaced by label names in both ALGOL and PL/1 . For example, ZL1ØØ could be used to replace 1ØØ . This change can be brought about by updating the subroutine DOLLAR (contained in SALIB).

5. CONTINUE and <label number> CONTINUE

These should be replaced by ; and ZL <label number> ; in both ALGOL and PL/1 .

6. GOTO - - -

This should be replaced by 'GOTO' in ALGOL.

7. Conditional statements

For example, the statement

```
        IF (X.EQ.Y)  GOTO  1ØØ
```

should be replaced by

```
        'IF' (X.EQ.Y)  'THEN'  'GOTO'  ZL1ØØ ;              (ALGOL)

   or   IF  X = Y  THEN GOTO ZL1ØØ ;                         (PL/1)
```

The forms .EQ. , .LE. , .LT. , .GT. , .GE. , .NE. , are replaced by = , < = , < , > , > = , ¬= respectively in PL/1.

8. <LHS> = <RHS>

should be replaced by

```
    <LHS> :  = <RHS> ;          (ALGOL)
    <LHS>    = <RHS> ;          (PL/1)
```

9. CALL - - -

CALL should not appear in ALGOL subroutine calls.

10. Label parameters

These are used in some FORTRAN subroutines, for example

```
CALL GVOVER (1, &1)
```

The appropriate forms should be used in ALGOL and PL/1 .

11. Switch statements

These exist in macro numbers 75 and 33. These macros should be updated to the forms required by either ALGOL or PL/1 .

APPENDIX 8

(1) THE FUNCTION OF RUN-TIME GLOBAL VARIABLES

A.	INTEGER VARIABLES		page
1.	ZTEMP	An array where the elements of non-scalar integer results are stored	137
2.	ZCBNDS	An array where the current bounds for an expression are stored	88
3.	ZCPTR	The pointer for array ZCBNDS.	88
4.	ZY	An array where successive base levels of ZTEMP are stored.	137
5.	ZYPTR	The pointer for array ZY	137
6.	ZYY	An array where successive base levels of YTEMP (see B) are stored.	137
7.	ZYYPTR	The pointer for array ZYY	137
8.	ZBONDS	An array where the bounds for <u>all</u> arrays are stored consecutively.	8
9.	ZPOINT	An array where successive base levels of ZINDX (see 11) are stored	17
10.	ZPT	The pointer for array ZPOINT.	17
11.	ZINDX	An array where the subscript values are stored during array accesses.	17
12.	ZBOOL	An array used to indicate previous occurrences of a random number (used in handling the "deal" function).	143

			page
13.	ZCOORD	An array where co-ordinate values (specified for mixed functions) are stored	96
14.	ZCDPTR	The pointer for array ZCOORD	96
15.	ZSTORE	An array containing link information for the blocks of YSTORE (see B)	8
16.	ZROWNO	The pointer for the array YROWL	135
17.	ZROWNA	The pointer for the array YROWR	135
18.	ZBPTR	The pointer to the next free location of ZBONDS	12
19.	ZØ	A name generated when an indexed expression occurs	18
20.	ZB1,ZB2,...	Locally generated scalars	18
21.	Z ---	Integer variable names	40
22.	MARKER	Gives type of result	89
23.	DOPES	An array for the dope vector table	8
24.	NAMES	An array for identifier names	10
25.	ZL	An array used to store elements of literals when quote-quad input is used	
26.	ZROW	An array where the left operand elements are stored when dyadic "rho" is handled	
27.	ZGRAD	An array where the result vector for "grade-up" and "grade-down" is stored	
28.	ZDIM	An array where the co-ordinate values for a particular row of an array are stored (used in handling n-dimensional accessing)	

page

29. ZSUB An array where subscript information is stored during accessing of n-dimensional arrays

30. ZTEMP3 An array where subscript information is stored for n-dimensional accessing in inner products

B. <u>REAL VARIABLES</u>

1. YTEMP An array where the elements of non-scalar real results are stored 137

2. YSTORE An array used for storage of all numeric non-scalar elements 7

3. YROWL An array used for storage of the left operand elements for certain mixed functions 135

4. YROWR An array used for storage of the right operand elements of certain mixed functions 135

5. Y --- Real variable names 40

6. YBOUND An array used for auxiliary storage of non-scalar elements

7. YBOND An array used for auxiliary storage of non-scalar elements

8. YGRAD An array used for storage of operand elements for "grade-up" and "grade-down" functions

9. YTEMP2 An array used for auxiliary storage of dyadic "rho" right operands

10. YTEM An array used to store numeric information supplied in response to "quad-input"

11. XBOUND An array used for auxiliary storage of non-scalar elements

(2) THE FUNCTION OF TRANSLATION TIME VARIABLES

			page
1.	FNIND	An array of variable name indices	55
2.	FNLOCS	An array of "locking" indices	53
3.	FNPARM	An array of indices of parameter names	58
4.	IBIT	The pointer for array IBITS	86
5.	IBITS	A character stack	86
6.	IBPTR	The pointer for array ITEMP	86
7.	ICLPTR	A pointer value from MLTAB	119
8.	ICOLM	The current value of ICLPTR, retained when the macro instruction RCM is used	123
9.	IDLPTR	The pointer for the chain on the stack IDSTK	79
10.	IDOLR	Current label number value	126
11.	IOPTR	A pointer for IDSTK	78
12.	IDSTK	A double-ended stack for operands and operators	77
13.	IEXP	Used to distinguish function and subroutine definition header statements	70
14.	IFIND	A stack of 1st parameter values for FIND calls	98
15.	IFNI	Gives the number of variable names in a function or subroutine definition header statement	70
16.	IFNPTR	The pointer for stack IFIND	98
17.	IFUNCT	Used in decoding function definitions	70
18.	IHET	Used to denote a heterogeneous output statement	95
19.	ILEFT	Used to indicate the presence of [	89
20.	IOPTON	Used to indicate the output medium required	24
21.	IOPTR	A pointer for the stack IDSTK	78
22.	IREV	A marker	113
23.	ISYMBT	The symbol table array of characters	36
24.	ITEMP	An intermediate result code array	86

			page
25.	LCHAR	Gives value of next character	60
26.	LINE	A character array for storage of input lines	6
27.	LITBLE	A 2-dimensional table with entries for literal variables	46
28.	LOCAL	An array of local variable name indices	54
29.	LOCS	An array giving the number of entries to be removed from LOCAL	54
30.	LPTR	The left pointer for NCODE	60
31.	LTABLE	A 2-dimensional table with entries for labels	154
32.	MACROS	An array of macro records	109
33.	MCADDR	An array of start addresses for macro bodies	33
34.	MLTAB	A 2-dimensional table for macro labels	119
35.	MTEMP	A temporary output array	86
36.	NADDR	Contains address in ISYMBT of current symbol decoded	36
37.	NAME	An array giving the current identifier name	37
38.	NCODE	Array of intermediate code	59
39.	NCOORD	Gives count of [symbols	96
40.	NEXP	Array used to distinguish functions and subroutines	56
41.	NLEFT	A marker	113
42.	NOLINE	A character array containing a processed APL line	32
43.	NOLPTR	Gives the number of entries in a particular APL line	35
44.	RPTR	The right pointer for NCODE	60
45.	SS	A looping array for repetition of macro statements	123
46.	TEMPR	The pointer for array MTEMP	110
47.	ZMARK	A parameter (0 to 5)	115

APPENDIX 9

EXAMPLES OF CONVERTED ROUTINES

Three examples are given illustrating the conversion of APL to FORTRAN by the method described.

As indicated in Chapter I, two possibilities existed for the handling of global variables during conversion.

1. Global variables could have been inserted in the parameter list, making parameter linkage a costly operation with regard both to space and execution time.

2. Global variables could have been inserted in the COMMON list, reducing the amount of parameter linkage required.

Conversion of APL to other languages instead of FORTRAN is made more difficult if method 2 is used.

Method 1 is used in the sample translations listed below.

In each of the examples listed, a number of non-executable statements appear. Some of these statements indicate the type and the number of dimensions of the global variables. In addition, the COMMON statements

```
        COMMON/ C7Ø1 / ZLIM1
                to
        COMMON/ C728 / ZLIM28
```

are present. The variables ZLIM1 to ZLIM28 are limit variables for certain global non-scalars. Their values are set on execution of the converted routines and they are used to test for overflow of global non-scalars.

For any routine, it is not known at the start of code production which of the variables ZLIM1 to ZLIM28 will be required. For this reason the complete list has to be inserted.

No optimisation of code is attempted at code generation stage, and thus the code produced is often inefficient. However, significant increases in efficiency are possible using the methods outlined in Chapter VIII.

The resultant FORTRAN code produced is necessarily more wieldy than the original APL code, as the generality of APL has to be catered for.

EXAMPLE 1

This simple example illustrates the conversion of a routine which calculates the surface area and volume of a sphere, given the radius R .

```
∇ SPHERE

  SURF ← 4 x 3.14159 x R x R

  VOL ← SURF x R ÷ 3  ∇
```

The following code is generated corresponding to the above routine.

```
 SUBROUTINE SPHERE(ZTEMP,YTEMP,YSTORE,YROWL,YROWR,ZCBNDS,ZCPTR,ZY,Z
CYPTR,ZYY,ZYYPTR,ZBONDS,YBOUND,YBOND,  ZPOINT,ZPT,ZINDX,ZBOOL,ZCOORD,
CZCDPTR,ZSTORE,ZL,YGRAD,ZROW,ZGRAD,ZDIM,YTEMP2,YTEM,XBOUND,ZTEMP3)
      IMPLICIT REAL(A-Y)
      IMPLICIT INTEGER(Z-Z)
      REAL YSTORE(1),YTEMP(1)
      REAL YTEMP2(1)
      REAL YTEM(1)
      REAL YGRAD(1)
      LOGICAL*1 MCHAC(1)
      REAL XBOUND(1)
      REAL YBOUND(1)
      REAL YBOND(1)
      LOGICAL*1 ZL(1),NAMES(5ØØØ),ZBOOL(1)
      LOGICAL*1 ZTYPEL(4)
      REAL YROWR(1)
      REAL YROWL(1)
      INTEGER ZTEMP(1)
      INTEGER ZTEMP3(1)
      INTEGER ZY(1)
      INTEGER ZYY(1)
      INTEGER ZCBNDS(1)
      INTEGER ZCOORD(1)
      INTEGER ZINDX(1),ZPOINT(1),ZBOUND(1Ø)
      INTEGER ZBONDS(1)
      INTEGER INAMES(125Ø),INAME(75)
      INTEGER DOPES(64,6),ZSTORE(1ØØ)
      INTEGER ZROW(1)
      INTEGER ZGRAD(1)
      INTEGER ZDIM(1),ZSUB(1)
      COMMON/ C24/ZBPTR
      COMMON/ C62/ZROWNO
      COMMON/ C63/ZROWNA
      COMMON/ C3/INAMES,INAME,KEY
      COMMON/ C2/ZSPACE,IADRES,DOPES
      COMMON/ C916/ZSAVE
      COMMON/ C351/MARKER
      COMMON/ C7Ø1/ZLIM1
      COMMON/ C7Ø2/ZLIM2
      COMMON/ C7Ø3/ZLIM3
      COMMON/ C7Ø4/ZLIM4
      COMMON/ C7Ø5/ZLIM5
      COMMON/ C7Ø6/ZLIM6
      COMMON/ C7Ø7/ZLIM7
      COMMON/ C7Ø8/ZLIM8
      COMMON/ C7Ø9/ZLIM9
      COMMON/ C71Ø/ZLIM1Ø
      COMMON/ C711/ZLIM11
      COMMON/ C712/ZLIM12
      COMMON/ C713/ZLIM13
      COMMON/ C714/ZLIM14
      COMMON/ C715/ZLIM15
      COMMON/ C716/ZLIM16
      COMMON/ C717/ZLIM17
      COMMON/ C718/ZLIM18
      COMMON/ C719/ZLIM19
      COMMON/ C72Ø/ZLIM2Ø
      COMMON/ C721/ZLIM21
      COMMON/ C722/ZLIM22
      COMMON/ C723/ZLIM23
      COMMON/ C724/ZLIM24
      COMMON/ C725/ZLIM25
      COMMON/ C726/ZLIM26
      COMMON/ C727/ZLIM27
      COMMON/ C728/ZLIM28
      COMMON/ C799/ZSTOP
      EQUIVALENCE(INAMES,NAMES)
      EQUIVALENCE(ZTYPE,ZTYPEL)
1     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      Y2=4*(3.14159*(R*R))
      IF(MARKER .NE.Ø) GOTO 111
      SURF=Y2
      GOTO 117
111   CALL SPECS(9,Y2,Z2,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZST
     CORE,ZCOORD,ZCDPTR,YBOUND)
      CALL SPECB
117   CONTINUE
2     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      Y3=SURF*(R/3)
      IF(MARKER .NE.Ø)  GOTO 123
      VOL=Y3
      GOTO 129
123   CALL SPECS(33,Y3,Z3,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZST
     CORE,ZCOORD,ZCDPTR,YBOUND)
129   CONTINUE
1ØØØ  IF(Z1.LE.Ø.OR.Z1.GT.3) CONTINUE
      GOTO (1,2,3),Z1
3     CONTINUE
1Ø1   FORMAT(1X,G12.6)
1Ø2   FORMAT(G12)
1Ø3   FORMAT(1X,I12)
1Ø4   FORMAT(1ØG12.6)
1Ø5   FORMAT(8ØA1)
1Ø6   FORMAT(1X,/)
      CALL LOCREM
      RETURN
      END
```

EXAMPLE 2

This example illustrates the handling of APL input and output expressions.

```
∇ CI ; A ; I ; Y

    'ENTER CAPITAL AMOUNT IN DOLLARS'

    A ← ⎕

    'ENTER' INTEREST IN PERCENT'

    I ← ⎕

    'ENTER PERIOD IN YEARS'

    Y ← ⎕

    'RESULT IS' ; A x (1 + .01 x I) * Y   ∇
```

The following code was generated corresponding to the above routine.

```
 SUBROUTINE CI(ZTEMP,YTEMP,YSTORE,YROWL,YROWR,ZCBNDS,ZCPTR,ZY,ZYPTR
C,ZYY,ZYYPTR,ZBONDS,YBOUND,YBOND,ZPOINT,ZPT,ZINDX,ZBOOL,ZCOORD,ZCDP
CTR,ZSTORE,ZL,YGRAD,ZROW,ZGRAD,ZDIM,ZSUB,YTEMP2,YTEM,XBOUND,ZTEMP3)
      IMPLICIT REAL(A-Y)
      IMPLICIT INTEGER(Z-Z)
      REAL YSTORE(1),YTEMP(1)
      REAL YTEMP2(1)
      REAL YTEM(1)
      REAL YGRAD(1)
      LOGICAL*1 MCHAC(1)
      REAL XBOUND(1)
      REAL YBOUND(1)
      REAL YBOND(1)
      LOGICAL*1 ZL(1),NAMES(5ØØØ),ZBOOL(1)
      LOGICAL*1 ZTYPEL(4)
      REAL YROWR(1)
      REAL YROWL(1)
      INTEGER ZTEMP(1)
      INTEGER ZTEMP3(1)
      INTEGER ZY(1)
      INTEGER ZYY(1)
      INTEGER ZCBNDS(1)
      INTEGER ZCOORD(1)
      INTEGER ZINDX(1),ZPOINT(1),ZBOUND(1Ø)
      INTEGER ZBONDS(1)
      INTEGER INAMES(125Ø),INAME(75)
      INTEGER DOPES(64,6),ZSTORE(1ØØ)
      INTEGER ZROW(1)
      INTEGER ZGRAD(1)
      INTEGER ZDIM(1),ZSUB(1)
      COMMON /C24/ZBPTR
      COMMON /C62/ZROWNO
      COMMON /C63/ZROWNA
      COMMON /C3/INAMES,INAME,KEY
      COMMON /C2/ZSPACE,IADRES,DOPES
      COMMON /C916/ZSAVE
      COMMON /C351/MARKER
      COMMON /C7Ø1/ZLIM1
      COMMON /C7Ø2/ZLIM2
      COMMON /C7Ø3/ZLIM3
      COMMON /C7Ø4/ZLIM4
      COMMON /C7Ø5/ZLIM5
      COMMON /C7Ø6/ZLIM6
      COMMON /C7Ø7/ZLIM7
      COMMON /C7Ø8/ZLIM8
      COMMON /C7Ø9/ZLIM9
      COMMON /C71Ø/ZLIM1Ø
      COMMON /C711/ZLIM11
      COMMON /C712/ZLIM12
      COMMON /C713/ZLIM13
      COMMON /C714/ZLIM14
      COMMON /C715/ZLIM15
      COMMON /C716/ZLIM16
      COMMON /C717/ZLIM17
      COMMON /C718/ZLIM18
      COMMON /C719/ZLIM19
      COMMON /C72Ø/ZLIM2Ø
      COMMON /C721/ZLIM21
      COMMON /C722/ZLIM22
      COMMON /C723/ZLIM23
      COMMON /C724/ZLIM24
      COMMON /C725/ZLIM25
      COMMON /C726/ZLIM26
      COMMON /C727/ZLIM27
      COMMON /C728/ZLIM28
      COMMON /C799/ZSTOP
      EQUIVALENCE(INAMES,NAMES)
      EQUIVALENCE(ZTYPE,ZTYPEL)
1     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      CALL OUT2(15,-1,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,ZDIM,ZSUB,YSTORE)
2     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      READ 1Ø2,ZVBND
      IF(ZVBND.NE.-1)  GOTO 116
      WRITE(6,1Ø6)
      MARKER=-3
      GOTO 12Ø
116   IF(ZVBND.NE.Ø)  GOTO 118
      MARKER=Ø
      READ(5,1Ø4) YTEM(1)
      WRITE(6,1Ø4) YTEM(1)
      GOTO 12Ø
118   IF(ZVBND.GT.ZLIM26) CALL GVOVER(26,&12Ø)
      READ(5,1Ø4)  (YTEM(Z2),Z2=1,ZVBND)
      WRITE(6,1Ø4)  (YTEM(Z2),Z2=1,ZVBND)
      ZCPTR=1
      ZCBNDS(1)=ZVBND
      MARKER=-5
12Ø   Z3=Ø
```

```
121  Z3=Z3+1
     Y4=YTEM(Z3)
     IF(MARKER.NE.Ø)  GOTO 122
     A=Y4
     GOTO 128
122  CALL SPECS(5,Y4,Z3,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZST
    CORE,ZCOORD,ZCDPTR,YBOUND)
     CALL BDNO(Z5,ZCPTR)
     IF(Z3.LT.Z5) GOTO 121
     MARKER=Ø
     ZCPTR=Ø
     ZYPTR=1
     ZYYPTR=1
     CALL SPECB
128  CONTINUE
3    CONTINUE
     MARKER=Ø
     ZCPTR=Ø
     CALL OUT2(54,-1,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,ZDIM,ZSUB,YSTORE)
4    CONTINUE
     MARKER=Ø
     ZCPTR=Ø
     READ 1Ø2,ZVBND
     IF(ZVBND.NE.-1) GOTO 139
     WRITE(6,1Ø6)
     MARKER=-3
     GOTO 143
139  IF(ZVBND.NE.Ø)  GOTO 141
     MARKER=Ø
     READ(5,1Ø4) YTEM(1)
     WRITE(6,1Ø4) YTEM(1)
     GOTO 143
141  IF(ZVBND.GT.ZLIM26) CALL GVOVER(26,&143)
     READ(5,1Ø4)  (YTEM(Z6),Z6=1,ZVBND)
     WRITE(6,1Ø4)  (YTEM(Z6),Z6=1,ZVBND)
     ZCPTR=1
     ZCBNDS(1)=ZVBND
     MARKER=-5
143  Z7=Ø
144  Z7=Z7+1
     Y8=YTEM(Z7)
     IF(MARKER.NE.Ø)  GOTO 145
     I=Y8
     GOTO 151
145  CALL SPECS(8,Y8,Z7,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZST
    CORE,ZCOORD,ZCDPTR,YBOUND)
     CALL BDNO(Z9,ZCPTR)
     IF(Z7.LT.Z9) GOTO 144
     MARKER=Ø
     ZCPTR=Ø
     ZYPTR=1
     ZYYPTR=1
     CALL SPECB
151  CONTINUE
5    CONTINUE
     MARKER=Ø
     ZCPTR=Ø
     CALL OUT2(87,-1,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,ZDIM,ZSUB,YSTORE)
6    CONTINUE
     MARKER=Ø
     ZCPTR=Ø
     READ 1Ø2,ZVBND
```

```
      IF(ZVBND.NE.-1) GOTO 162
      WRITE(6,1Ø6)
      MARKER=-3
      GOTO 166
162   IF(ZVBND.NE.Ø) GOTO 164
      MARKER=Ø
      READ(5,1Ø4) YTEM(1)
      WRITE(6,1Ø4) YTEM(1)
      GOTO 166
164   IF(ZVBND.GT.ZLIM26) CALL GVOVER(26,&166)
      READ(5,1Ø4) (YTEM(Z1Ø),Z1Ø=1,ZVBND)
      WRITE(6,1Ø4) (YTEM(Z1Ø),Z1Ø=1,ZVBND)
      ZCPTR=1
      ZCBNDS(1)=ZVEND
      MARKER=-5
166   Z11=Ø
167   Z11=Z11+1
      Y12=YTEM(Z11)
      IF(MARKER.NE.Ø) GOTO 168
      YY=Y12
      GOTO 174
168   CALL SPECS(11,Y12,Z11,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,
      CZSTORE,ZCOORD,ZCDPTR,YBOUND)
      CALL BDNO(Z13,ZCPTR)
      IF(Z11.LT.Z13) GOTO 167
      MARKER=Ø
      ZCPTR=Ø
      ZYPTR=1
      ZYYPTR=1
      CALL SPECB
174   CONTINUE
7     CALL OUT2(116,-1,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,ZDIM,ZSUB,YSTORE)
      IF(MARKER.NE.Ø) GOTO 180
      Y14=A*((1+(Ø.1*I))**YY)
      WRITE(6,1Ø1) Y14
      GOTO 186
18Ø   Y15=A*((1+(Ø.1*I))**YY)
      CALL OUT1(Y15,ZCBNDS,ZCPTR)
186   CONTINUE
1ØØØ IF(Z1.LE.Ø.OR.Z1.GT.7) CONTINUE
      GOTO(1,2,3,4,5,6,7),Z1
1Ø1   FORMAT(1X,G12,6)
1Ø2   FORMAT(G12)
1Ø3   FORMAT(1X,I12)
1Ø4   FORMAT(1ØG12.6)
1Ø5   FORMAT(8ØA1)
1Ø6   FORMAT(1X,/)
      CALL LOCREM
      RETURN
      END
```

EXAMPLE 3

This example illustrates the handling of non-scalar variables. The routine BASE calculates the representation of a number N to the base B. N and B are supplied as parameters.

```
    ∇ B  BASE  N
    Z ← ι∅
    R ← B|N
    Z ← R , Z
    N ← ⌊N ÷ B
  → 2 x N > ∅ ∇
```

The following code is generated corresponding to the above routine.

```
 SUBROUTINE BASE(B,N,ZTEMP,YTEMP,YSTORE,YROWL,YROWR,ZCBNDS,ZCPTR,ZY
C,ZYPTR,ZYY,ZYYPTR,ZBONDS,YBOUND,YBOND,ZPOINT,ZPT,ZINDX,ZBOOL,ZCOOR
CD,ZCDPTR,ZSTORE,ZL,YGRAD,ZROW,ZGRAD,ZDIM,ZSUB,YTEMP2,YTEM,XBOUND,Z
CTEMP3)
      IMPLICIT REAL(A-Y)
      IMPLICIT INTEGER(Z-Z)
      REAL YSTORE(1),YTEMP(1)
      REAL YTEMP2(1)
      REAL YTEM(1)
      REAL YGRAD(1)
      LOGICAL*1 MCHAC(1)
      REAL XBOUND(1)
      REAL YBOUND(1)
      REAL YBOND(1)
      LOGICAL*1 ZL(1),NAMES(5ØØØ),ZBOOL(1)
      LOGICAL*1 ZTYPEL(4)
      REAL YROWR(1)
      REAL YROWL(1)
      INTEGER ZTEMP(1)
      INTEGER ZTEMP3(1)
      INTEGER ZY(1)
      INTEGER ZYY(1)
      INTEGER ZCBNDS(1)
      INTEGER ZCOORD(1)
      INTEGER ZINDX(1),ZPOINT(1),ZBOUND(1Ø)
      INTEGER ZBONDS(1)
      INTEGER INAMES(125Ø),INAME(75)
      INTEGER DOPES(64,6),ZSTORE(1ØØ)
      INTEGER ZROW(1)
      INTEGER ZGRAD(1)
      INTEGER ZDIM(1),ZSUB(1)
      COMMON /C24/ZBPTR
      COMMON /C62/ZROWNO
      COMMON /C63/ZROWNA
      COMMON /C3/INAMES,INAME,KEY
      COMMON /C2/ZSPACE,IADRES,DOPES
      COMMON /C916/ZSAVE
      COMMON /C351/MARKER
      COMMON /C7Ø1/ZLIM1
      COMMON /C7Ø2/ZLIM2
      COMMON /C7Ø3/ZLIM3
      COMMON /C7Ø4/ZLIM4
      COMMON /C7Ø5/ZLIM5
      COMMON /C7Ø6/ZLIM6
      COMMON /C7Ø7/ZLIM7
      COMMON /C7Ø8/ZLIM8
      COMMON /C7Ø9/ZLIM9
      COMMON /C71Ø/ZLIM1Ø
      COMMON /C711/ZLIM11
      COMMON /C712/ZLIM12
      COMMON /C713/ZLIM13
      COMMON /C714/ZLIM14
      COMMON /C715/ZLIM15
      COMMON /C716/ZLIM16
      COMMON /C717/ZLIM17
      COMMON /C718/ZLIM18
      COMMON /C719/ZLIM19
      COMMON /C72Ø/ZLIM2Ø
      COMMON /C721/ZLIM21
      COMMON /C722/ZLIM22
      COMMON /C723/ZLIM23
      COMMON /C724/ZLIM24
      COMMON /C725/ZLIM25
      COMMON /C726/ZLIM26
      COMMON /C727/ZLIM27
      COMMON /C728/ZLIM28
      COMMON /C799/ZSTOP
      EQUIVALENCE(INAMES,NAMES)
      EQUIVALENCE(ZTYPE,ZTYPEL)
1     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      Y2=Ø
      IF(Y2.NE.Ø) GOTO 111
      MARKER=-3
      ZCPTR=Ø
      GOTO 112
111   CALL JOTB(Y2,ZTEMP,ZY,ZYPTR,ZCBNDS,ZCPTR)
      Z3=ZY(ZYPTR-1)
      ZY(ZYPTR)=Ø
      ZYPTR=ZYPTR-1
      MARKER=-5
112   Z4=Ø
113   Z4=Z4+1
      Y5=ZTEMP(Z4+Z3)
      IF(MARKER.NE.Ø) GOTO 114
      YZ=Y5
      GOTO 12Ø
114   CALL SPECS(1,Y5,Z4,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,ZST
     CORE,ZCOORD,ZCDPTR,YBOUND)
      CALL BDNO(Z6,ZCPTR)
      IF(Z4.LT.Z6) GOTO 113
      MARKER=Ø
      ZCPTR=Ø
      ZYPTR=1
      ZYYPTR=1
      CALL SPECB
12Ø   CONTINUE
2     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      Y7=B
      Y8=N
      Y9=Y8
      IF(Y7.NE.Ø)Y9=Y9-ABS(Y7)*AINT(Y8/ABS(Y7))
      Y1Ø=Y9
      IF(MARKER.NE.Ø) GOTO 126
      R=Y1Ø
      GOTO 132
126   CALL SPECS(23,Y1Ø,Z1Ø,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,
     CZSTORE,ZCOORD,ZCDPTR,YBOUND)
      CALL SPECB
132   CONTINUE
3     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      ZBØ=ZPOINT(ZPT)
      ZPT=ZPT+1
      CALL STARTS(1,Z11,Z12,ZNC,ZCPTR,ZCBNDS)
      ZPOINT(ZPT)=ZBØ+Z12
      Z13=1
138   Z14=ZBØ+Z13
      ZINDX(Z14)=1
      Z13=Z13+1
      IF(Z13.LE.Z12) GOTO 138
      Z15=ZBØ+Z12
      Z16=Z12-1
      ZSAVE=Ø
137   CALL FIND1(14,Ø,R,1,Z17,Y17,Z11,ZNC,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT
     C ,YBOUND,ZINDX,ZSTORE,ZCOORD,ZCDPTR,YSTORE)
      Y18=Y17
      CALL SPECS(1,Y18,Z18,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,Z
     CSTORE,ZCOORD,ZCDPTR,YBOUND)
      ZSAVE=1
      ZINDX(Z15)=ZINDX(Z15)+1
      IF(ZINDX(Z15).LE.ZCBNDS(ZCPTR)) GOTO 137
139   Z19=Z16+1
14Ø   ZINDX(Z19+ZBØ)=1
      Z19=Z19+1
      IF(Z19.LE.ZCPTR) GOTO 14Ø
141   IF((ZBØ+Z16).LE.Ø) GOTO 143
      ZINDX(ZBØ+Z16)=ZINDX(ZBØ+Z16)+1
      IF(ZINDX(ZBØ+Z16).LE.ZCBNDS(Z16)) GOTO 142
      IF(Z16.EQ.1) GOTO 143
      ZINDX(ZBØ+Z16)=1
      Z16=Z16-1
      GOTO 141
142   Z16=Z12-1
      GOTO 137
143   ZPT=ZPT-1
      MARKER=Ø
      ZCPTR=Ø
      CALL SPECB
144   CONTINUE
4     CONTINUE
```

```
      MARKER=Ø
      ZCPTR=Ø
      Y2Ø=N/B
      IF(Y2Ø.GE.Ø) GOTO 15Ø
      Y2Ø=ABS(Y2Ø-(1-1E-8))
15Ø   Y20=AINT(Y2Ø)
      Y21=Y2Ø
      IF(MARKER.NE.Ø) GOTO 152
      N=Y21
      GOTO 158
152   CALL SPECS(14,Y21,Z21,ZCBNDS,ZCPTR,ZBONDS,ZPOINT,ZPT,YSTORE,ZINDX,
     CZSTORE,ZCOORD,ZCDPTR,YBOUND)
      CALL SPECB
158   CONTINUE
5     CONTINUE
      MARKER=Ø
      ZCPTR=Ø
      CALL OUT2(1,1,ZCBNDS,ZCPTR,ZBONDS,ZSTORE,ZDIM,ZSUB,YSTORE)
6     CONTINUE
      Z22=Ø
      IF(N.GT.Ø)Z22=1
      IF(MARKER.EQ.-3) GOTO 169
      IF(MARKER.NE.Ø) GOTO 17Ø
      Z1=2*Z22
      GOTO 1ØØØ
17Ø   Z22=1
      Z1=2*Z22
      GOTO 1ØØØ
169   CONTINUE
1ØØØ  IF(Z1.LE.Ø.OR.Z1.GT.6) CONTINUE
      GOTO(1,2,3,4,5,6),Z1
1Ø1   FORMAT(1X,G12.6)
1Ø2   FORMAT(G12)
1Ø3   FORMAT(1X,I12)
1Ø4   FORMAT(1ØG12.6)
1Ø5   FORMAT(8ØA1)
1Ø6   FORMAT(1X,/)
      CALL LOCREM
      RETURN
      END
```

APPENDIX 10

This appendix describes the finite state automaton which may be implemented for replacement of unnecessary "FIND" calls. The method is outlined in Chapter VIII, §8.1 and §8.2 .

Table 1Ø(a) gives a list of statement types to be recognised in the sçan of the generated code. For ease of reference in the state diagram, each statement type has an associated letter (or letters).

The state diagram is represented by Diagram 1Ø(b).

For statement types not listed in Table 1Ø(a), the action required is outlined in Chapter VIII, §8.2 .

STATEMENT TYPE	ASSOCIATED LETTER(S)
ZB <integer> = ZPOINT (ZPT)	A
ZPT = ZPT + 1	B
ZINDX (ZB < integer > + <integer 1>) = < expression >	C
ZPOINT (ZPT) = ZB <integer > +<integer 1>	D
CALL STARTS (- - - -)	E
CALL FIND1 (- - - -) (replaceable)	F
ZPT = ZPT - 1	G
Z <integer> = 1	H
<label> Z < integer 1> = ZB < integer 2> + Z<integer>	I
ZINDX (Z <integer 1>) = 1	J
Z<integer> = Z < integer> + 1	K
IF (Z < integer > .LE.Z <integer 3>) GOTO < label >	L
Z <integer> = ZB <integer 1 > + Z <integer 2 >	M

STATEMENT TYPE		ASSOCIATED LETTER(S)
Z<integer > = Z <integer 1> - 1		N
<label> CALL FIND1 (- - - -)	(replaceable)	O
CALL SPECS (- - - -)	(replaceable)	P
ZINDX (Z<integer>) = ZINDX (Z<integer>) + 1		Q
IF (ZINDX (Z<integer>).LE.ZCBNDS (ZCPTR)) GOTO <label>		R
<label> Z<integer 1> = Z <integer > + 1		S
<label> ZINDX (Z <integer> + ZB <integer>) = 1		T
IF (Z<integer>.LE.ZCPTR) GOTO <label>		U
<label> ZINDX (ZB <integer> + Z <integer 1>) = ZINDX (ZB <integer> + Z <integer 1>) + 1		V
IF (ZINDX (ZB<integer> + Z<integer 1>).LE. ZCBNDS (Z <integer 1>)) GOTO <label>		W
IF (Z < integer> .EQ.1) GOTO <label>		X
ZINDX (ZB <integer1> + Z <integer>) = 1		Y
Z <integer> = Z <integer > - 1		Z
GOTO < label >		AA
<label> Z <integer> = Z <integer> - 1		AB
<label> ZPT = ZPT - 1		AC
<label> Z <integer> = FIND (- - - -)	(replaceable)	AD
ZINDX (ZB <integer> + 1) = ZINDX (ZB< integer> + 1) + 1		AE
IF (ZINDX (ZB < integer>+1).LE.ZBONDS (ZBOUND)) GOTO <label >		AF
<label> ZB <integer > = ZPOINT (ZPT)		AG
ZINDX (ZCD + ZB <integer>) = ZINDX (ZCD+ZB<integer>) + 1		AH
IF(ZINDX (ZCD+ZB<integer>).LE.ZBONDS (ZBOUND+ZCD-1)) GOTO <label>		AI
IF(ZB < integer>+Z<integer 1> .LE.Ø) GOTO <label>		AJ
IF (Z<integer> .EQ.ZCD) Z <integer> = Z<integer> - 1		AK
CALL FIND1 (- - - -) CALL FIND1 (--- ZF1 ---) CALL FIND1 (--- ZF2 ---)	non-replaceable " "	AL

STATEMENT TYPE		ASSOCIATED LETTER(S)
<label> CALL FIND1 (- - -)	non-replaceable	AM
<label> CALL FIND1 (--- ZF1 ---)	"	
<label> CALL FIND1 (--- ZF2 ---)	"	
CALL SPECS (- - - -)	non-replaceable	AN
<label> Z <integer> = FIND(- - - -)	non-replaceable	AO
<label> <expression> = EVFIND(- - - -)	"	
<label> <expression> = SCFIND(- - - -)	"	
<expression> = IRFIND(- - - -)	"	
OPL = <value>		AP

TABLE 10(a) : Statement types to be detected during scan of generated code.

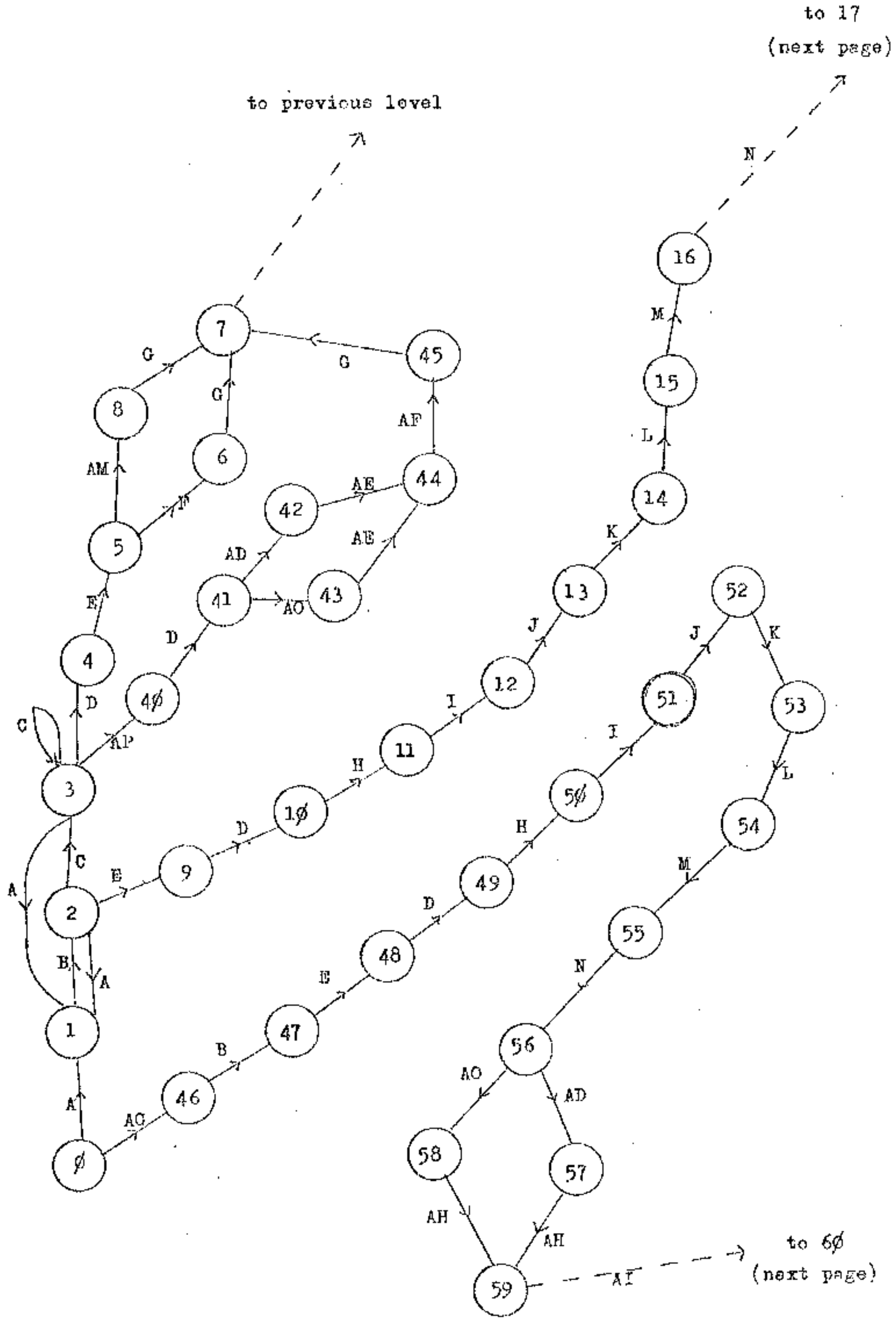


Diagram 1Ø(b) : Shows state diagram for replacement of unnecessary "FIND" calls.

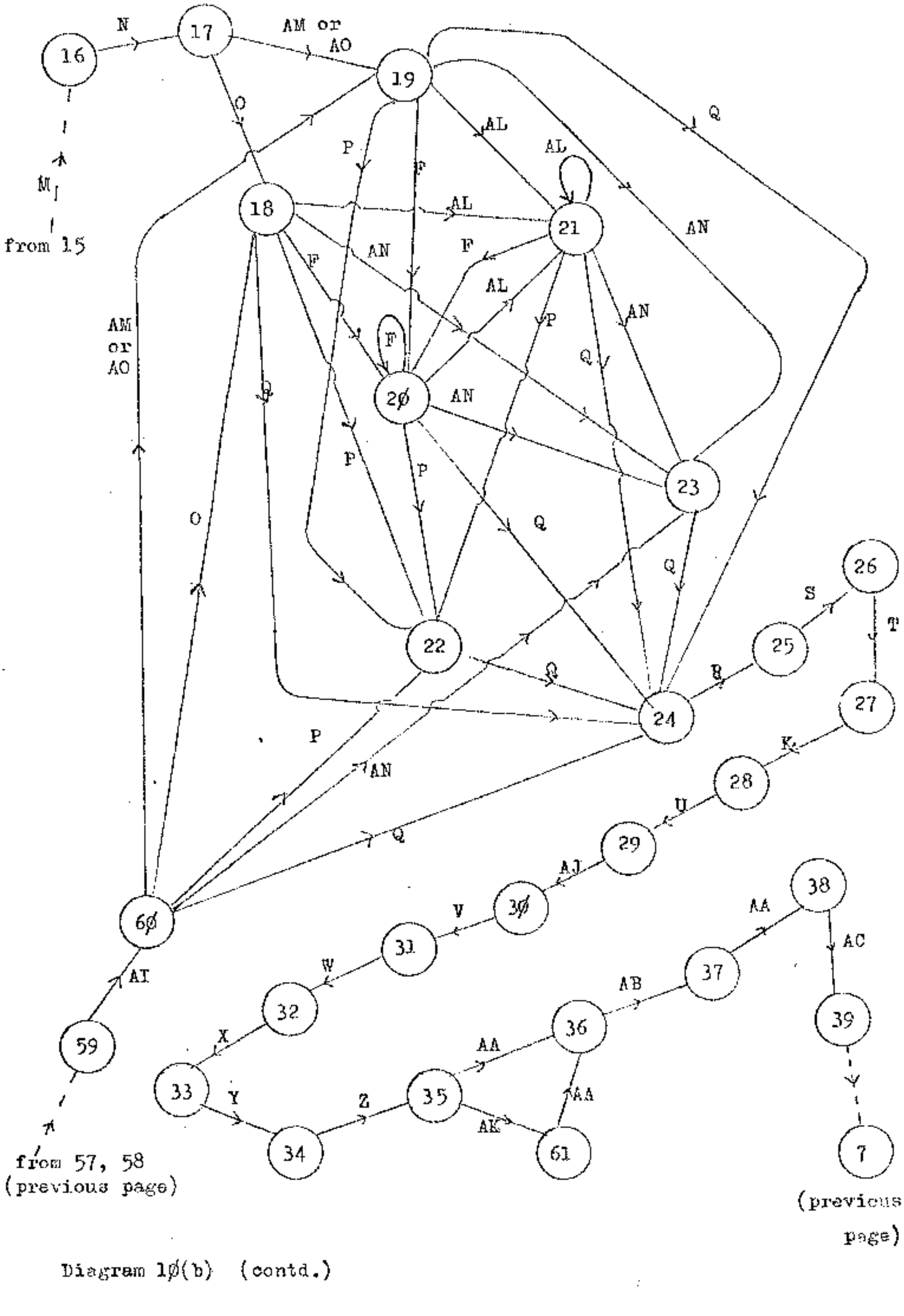


Diagram 1Ø(b) (contd.)

The following actions are required at each state represented in the state diagram.

State 1.

```
Set  NSTPTR to NSTPTR + 1
Set  NSTATE (NSTPTR) to ISTATE
Set  ISTATE to 1
Set LEVLNO to LEVLNO + 1
Set  IENPTR to IENPTR +1
Set  IENTRY ( IENPTR) to ∅
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

Store the value of DELPTR in a stack so that the replacement code for CALL SPECS (- - - -) may be inserted in the correct place.

State 2.

```
Set  ISTATE to 2
Set  DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to IENTRY (IENPTR)
```

State 3.

```
Set ISTATE to 3
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
Set  ICDPTR to ICDPTR + 1
Set  CODE (ICDPTR,1) to DELPTR
Set  CODE (ICDPTR,2) to 1
Set  CODE (ICDPTR,3) to -1
Set  CODE (ICDPTR,4) to point  to code of the form
```

```
Z(<LEVLNO value>, <integer1>)   =   <expression>
```

State 4.

```
Set  ISTATE to 4
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
Set  ICDPTR to ICDPTR + 1
```

A line of code of the form

```
ZPOINT(ZPT)  =  ZB <integer> + <integer1 >
```

has been recognised.

If <integer 1> is 1, then the following entry is set up in CODE

```
CODE (ICDPTR,1)   =   DELPTR
CODE (ICDPTR,2)   =   2
CODE (ICDPTR,3)   =  -1
CODE (ICDPTR,4)   =   a pointer to code of the form indicated below

CALL ELPERM (- - - -)
ZPROD   =   Z(<LEVLNO value>,1)
```

(The parameters of ELPERM are filled in later, if the replacement is to be made. Thus, an indication of the address of the ELPERM call must be retained.)

If <integer1> is greater than 1, then the following entry is set up in CODE.

```
        CODE (ICDPTR,1)   =   DELPTR
        CODE (ICDPTR,2)   =   3
        CODE (ICDPTR,3)   =  -1
        CODE (ICDPTR,4)   =   a pointer to code of the form indicated below.

        CALL ELPERM (- - - -
        Z<integer A >   =   <integer 1> -1
        ZPROD    =    Z(<LEVLNO value>, <integer 1>)
```

Again, the parameters of ELPERM must be filled in later (if necessary).

State 5

```
        Set  ISTATE to 5
        Set  DELPTR to DELPTR + 1
        Set  DELETE (DELPTR,1) to LEVLNO
        Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

If <integer 1 > (see state 4) is greater than 1, the following action is also required

```
        Set  ICDPTR to ICDPTR + 1
        Set  CODE (ICDPTR,1) to DELPTR
        Set  CODE (ICDPTR,2) to 5
        Set  CODE (ICDPTR,3) to -1
        Set CODE (ICDPTR, 4) to point to code of the following form

        CALL ZADDR (<value>,ZST,ZNUM,ZBOUND)
<label> IF (Z <integer A>.LE.1) GOTO <label 1 >
        ZPROD  =  (ZPROD-1)*ZBONDS (ZBOUND + Z <integer A> - 1)
        Z <integer A>  =  Z <integer A> - 1
        GOTO <label>
```

Here < value > is the 1st parameter value of the STARTS call.

For < integer 1> = 1, only the first line of the above code is required, and a corresponding CODE entry is set up.

State 6.

A replaceable FIND1 call has been detected at this stage. The DELETE table (column 1) must be scanned for entries equal in value to LEVLNO. For any such entry i, DELETE(i,2) is tested. If DELETE(i,2) is ∅, then it is updated to 1. The appropriate entries in CODE (column 3) should also be updated to 1 if they are -1 originally.

```
Then,   Set  ISTATE to 6
        Set  IENTRY (IENPTR) to 1
        Set  DELPTR to DELPTR + 1
        Set  DELETE (DELPTR,1) to LEVLNO
        Set  DELETE (DELPTR,2) to  1
        Set  ICDPTR to ICDPTR + 1
        Set  CODE (ICDPTR,1) to DELPTR
        Set  CODE (ICDPTR,2) to  1
        Set  CODE (ICDPTR,3) to  1
        Set  CODE (ICDPTR,4) to point to code of the form shown below
```

```
 <label 1> Y <integer B >  =   YSTORE(ZST + ZPROD - 1)
```

where <label 1> is retained from state 5 and Z <integer B > is the sixth parameter of the FIND1 call.

The parameters for the call of ELPERM can now be inserted.

State 7.

```
        Set  DELPTR to DELPTR + 1
        Set  DELETE (DELPTR,1) to LEVLNO
        Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
        Set  IENTRY (IENPTR) to  ∅
        Set  IENPTR to IENPTR - 1
```

```
Set  LEVLNO to LEVLNO - 1
Set  ISTATE to NSTATE (NSTPTR)
Set  NSTATE (NSTPTR) to -1
Set  NSTPTR to NSTPTR - 1
```

State 8.

A non-replaceable FIND1 call has been detected at this stage. The DELETE table (column 1) must be scanned for entries equal in value to LEVLNO. For any such entry **i**, if DELETE(i,2) is ∅, then DELETE(i,2) is set to -1. The appropriate CODE entries (column 3) should be updated to ∅ if they are -1 originally. The lines of (replacement) code produced for this level are not required, and the space can be utilised if required to produce more lines of code.

Then,

```
Set  ISTATE to 8
Set  IENTRY (IENPTR) to -1
Set  DELPTR  to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to -1
```

States 9-1∅.

```
Set  ISTATE to 9 or 1∅
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

State 11.

```
Set  ISTATE to 11
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

Store the value <integer> on a stack for use later (if the loops are replaceable).

State n $(12\le n\le 17)$

```
Set  ISTATE to  n
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

State 18

A replaceable FIND1 call has been detected. It is not yet known, however, whether the entire loops can be replaced. This will not become apparent until State 24 is reached.

```
Set  ISTATE to 18
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to 1
```

The <label> value in the statement detected should be stacked for (possible) use when state 24 is reached. Similarly, the value of DELPTR should be stacked. A count, C1, of the number of replaceable FIND or FIND1 calls should also be maintained until state 24 is reached.

The following CODE entry should be set up.

```
ICDPTR  =  ICDPTR + 1
CODE (ICDPTR,1)  =  DELPTR
CODE (ICDPTR,2)  =  3
CODE (ICDPTR,3)  =  1
CODE (ICDPTR,4)  =  a pointer to code of the form outlined below.

   Z <integer > =  1
   CALL ZADDR (<value>,ZST,ZNUM,ZBOUND)
   Z <integer 1> =  ZST - 1
```

Here < integer > has been retained from state 11, < value > is the fourth parameter of the FIND or FIND1 call, and Z < integer 1 > should be a unique variable name.

Z < integer 1> and the sixth parameter of the FIND1 call should be retained.

State 19

The same action should be carried out as for state 8, except that ISTATE is set to 19.

State 2∅

```
Set ISTATE to 2∅
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to 1
Set ICDPTR to ICDPTR + 1
Set CODE (ICDPTR,1) to DELPTR
Set CODE (ICDPTR,2) to 2
Set CODE (ICDPTR,3) to 1
Set CODE (ICDPTR,4) to point to code of the form

CALL ZADDR(<value>,ZST,ZNUM,ZBOUND)
Z <integer 1> = ZST - 1
```

< value> and <integer 1> have the same significance as in state 18. The same information should be retained here also.

State 21

The same action should be carried out as for state 8, except that ISTATE is set to 21.

If count C2 is non-zero, then the following CODE entry is also set up.

```
ICDPTR = ICDPTR + 1
CODE (ICDPTR,1) = the value of DELPTR stacked at state 18
CODE (ICDPTR,2) = 2 * C2
CODE (ICDPTR,3) = 1
CODE (ICDPTR,4) = a pointer to code of the form

Z <integer 1> = Z <integer 1> + 1
CALL SPECA (Z <integer 1>,<variable>)
```

The above lines are repeated C2 times, using the value <integer 1> retained previously. <value> is the left-hand side of the statement preceding the SPECS call.

State n (25≤n≤38)

```
Set ISTATE to n
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to IENTRY (IENPTR)
```

State 39

```
Set ISTATE to 39
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to IENTRY (IENPTR)
```

If IENTRY (IENPTR) is -1, proceed to state 7, otherwise set ICDPTR to ICDPTR + 1 .

```
Set CODE (ICDPTR,1) to DELPTR
Set CODE (ICDPTR,2) to 2
Set CODE (ICDPTR,3) to 1
Set CODE (ICDPTR,4) to point to code of the form shown below.
```

State 22

The same action is carried out as for state 2Ø, except ISTATE is set to 22, and CODE (ICDPTR,1) is set to the value of DELPTR stacked at state 1. Maintain a count, C2, of the number of replaceable "SPECS" calls encountered.

State 23

Set ISTATE to 23

Set DELPTR to DELPTR + 1

Set DELETE (DELPTR,1) to LEVLNO

Set DELETE (DELPTR,2) to -1

State 24

Set ISTATE to 24

Set DELPTR to DELPTR + 1

Set DELETE (DELPTR,1) to LEVLNO

Set DELETE (DELPTR,2) to IENTRY (IENPTR)

If the counts C1 and C2 are zero, continue the scan, otherwise set all the appropriate zero entries in DELETE to 1, update the relevant CODE entries and set up a new CODE entry as indicated below.

```
ICDPTR  =  ICDPTR + 1

CODE (ICDPTR,1)  =  the value of DELPTR stacked at state 18

CODE (ICDPTR,2)  =  2 * C1

CODE (ICDPTR,3)  =  1

CODE (ICDPTR,4)  =  a pointer to code of the form shown below.

Z <integer 1>  =  Z <integer 1> + 1
Y <integer C>  =  LPERM (Z <integer 1>,ZCBNDS,ZCPTR,I,J,K,L)
```

The above lines are repeated C1 times, using the values of < integer 1> and < integer C > retained previously. If state 18 was reached, the first statement is labelled < label> , where <label> is the value stacked at state 18.

```
Z <integer> = Z <integer> + 1
IF (Z <integer> .LE.ZNUM) GOTO <label>
```

Here < integer > was retained at state 11 and < label > at state 18.

Proceed to state 7.

State 4∅

```
Set ISTATE to 4∅
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to -1
```

Keep note of Z <integer C> at start of next line.

State 41

```
Set ISTATE to 41
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to IENTRY (IENPTR)
```

State 42

```
Set ISTATE to 42
Set DELPTR to DELPTR + 1
Set DELETE (DELPTR,1) to LEVLNO
Set DELETE (DELPTR,2) to 1
Set IENTRY (IENPTR) to 1
```

Scan the DELETE table (column 1) for entries having value LEVLNO. For any such entry i, if DELETE (i,2) is ∅, set DELETE (i,2) to 1 and update the appropriate CODE entry.

Set up a new CODE entry as indicated below.

```
ICDPTR = ICDPTR + 1
CODE (ICDPTR,1) = DELPTR
CODE (ICDPTR,2) = 4
CODE (ICDPTR,3) = 1
CODE (ICDPTR,4)  =  a pointer to code of the form given below.

<label> CALL ELPERM (---)
        ZPROD = Z (<LEVLNO value>,1)
        CALL ZADDR (<value>,ZST,ZNUM,ZBOUND)
        Z<integer>  =  ZST + ZPROD-1
```

Here, <label> is obtained from the statement detected and <value> is the fourth parameter of the FIND call.

In this case, the parameters for ELPERM can be inserted immediately.

<integer> is also obtained from the statement detected.

State 43

The action required is that described for state 8, except that ISTATE is set to 43.

State 44

```
Set  ISTATE to 44
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

If IENTRY (IENPTR) = -1, proceed to scan next statement. Otherwise set up a CODE entry of the following form:

```
ICDPTR  =  ICDPTR + 1
CODE (ICDPTR,1)  =  DELPTR
CODE (ICDPTR,2)  =  1
CODE (ICDPTR,3)  =  1
CODE (ICDPTR,4)  =  a pointer to CODE of the form shown below.

Z(<LEVLNO value>,1)  =  Z(<LEVLNO value>,1) + 1
```

State 45

```
Set  ISTATE to 45
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to IENTRY (IENPTR)
```

Set up a CODE entry of the following form if IENTRY (IENPTR) is not equal to -1.

```
ICDPTR  =  ICDPTR + 1
CODE (ICDPTR,1)  =  DELPTR
CODE (ICDPTR,2)  =  1
CODE (ICDPTR,3)  =  1
CODE (ICDPTR,4)  =  a pointer to code of the form shown below.
```

```
IF(Z < LEVLNO value>,1).LE.ZBONDS (ZBOUND))  GOTO <label>
```

Here <label> is obtained from the current statement detected.

State n $(46 \leq n \leq 56)$

```
Set  ISTATE to  n
Set  DELPTR to DELPTR + 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to -1
```

For state 46, stack the value of DELPTR so that, if necessary, the replacement CODE for CALL SPECS (----) can be inserted at the correct point.

State 57

```
Set  ISTATE to 57
Set  DELPTR to DELPTR + 1
Set  IENTRY (IENPTR) to 1
Set  DELETE (DELPTR,1) to LEVLNO
Set  DELETE (DELPTR,2) to 1
Set  ICDPTR to ICDPTR + 1
Set  CODE (ICDPTR,1) to DELPTR

Set  CODE (ICDPTR,2) to 4

Set  CODE (ICDPTR,3) to 1

Set  CODE (ICDPTR,4) to point to CODE of the form
```

```
          CALL ZADDR (<value>,ZST,ZNUM,ZBOUND)

          Z <integer A>  =   ZST - 1
< label>  Z <integer>    =   Z <integer C> + 1
          Z <integer B>  =   Z <integer A> + ZPERM (1,Ø,Ø,<value>,ZINDX,ZBCNDS,
                                             Z <integer>,ZPOINT,ZPT)
```

Here <value> is the fourth parameter of the FIND call, Z <integer A> is a unique variable name, <integer> is obtained from the current statement (as is <label>). Z <integer B> is another unique variable name and <integer C> was retained at state 4Ø .

<u>State 58</u>

```
Set  ISTATE to 58

Set  IENTRY (IENPTR) to -1

Set  DELPTR to DELPTR + 1

Set  DELETE (DELPTR,1) to LEVLNO

Set  DELETE (DELPTR,2) to -1
```

<u>State n</u> $(59 \leq n \leq 61)$

```
Set  ISTATE to  n

Set  DELPTR to DELPTR + 1

Set  DELETE (DELPTR,1) to LEVLNO

Set  DELETE (DELPTR,2) to -1
```

INDEX OF TERMS