THE UNIVERSITY OF ALBERTA

IMPROVEMENTS TO THE LANGUAGE AND IMPLEMENTATION OF APL

BY

FRED APPLEYARD

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

AND RESEARCH IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1973

ABSTRACT

Iverson's original APL differs substantially in some respects from implemented versions. In the course of implementing APL deficiencies have crept into the language in areas such as subscripting, branching, mixed operators and functions. The former part of this work is a review of these deficiencies as well as proposals to correct them.

The latter part of the thesis examines Abrams' work on an APL machine and proposes changes to it. The APL machine is a stack-oriented machine which can simplify APL expressions and defer their execution. Simplification is accomplished by means of transformations to the polynomial access functions of arrays. Deferral allows simplification to be carried out over larger expressions as well as optimizing the use of low speed core memory. The APL machine is more efficient than current implementations which immediately apply operators to their operands creating temporary results in low speed memory.

ACKNOWLEDGEMENT

I would like to express my appreciation to my initial supervisor, Professor W. S. Adams, for his guidance and criticism in preparing this thesis; to Dr. B. J. Mailloux for piloting me through the final stages while Professor Adams was absent on sabattical leave; and to the other members of my committee for the time and energy spent in improving the final form of this thesis. In addition, a note of thanks to Dr. D. B. Scott for the encouragement to pursue graduate studies and to Helen Pederson for rendering this work from an illegible scrawl to its present form.

TABLE OF CONTENTS

CHAPTER I

Introduction

The design and implementation of programming languages, such as APL(A Programming Language), is a compromise between design goals and pragmatic considerations in the implementation. Experience in implementing a language affects changes in the design of that language as well as influencing the design of new languages. This feedback mechanism is a major component in the development of programming languages. In discussing design criteria Iverson[14] states that a programming language should be:

> "concise, precise, consistent over a wide area of application, mnemonic, and economical of symbols; it should exhibit clearly constraints on the sequence in which operations are performed; and it should permit the description of a process to be independent of the particular representation chosen for the data."

Through the several implementations by L. M. Breed[8], P. S. Abrams[1], P. Berry[5], and others, many concessions have been made to ease the task of implementation. Although the results are readily recognizable as APL the design

criteria set forth by Iverson have not been rigorously applied to the alterations made. We will examine the language and implementation as described by the APL/360 User's Manual[12]. It is assumed that the reader has a reasonable working knowledge of the API language and programming. Appendix A contains a summary of the operators in the language.

The utility of a programming language is dependent to a significant degree on the consistency of its definition. APL is a simple and consistent language that has acquired some flaws in the process of implementing. Chapter II is concerned only with criticizing some of the inconsistencies of the implemented language. The criticisms deal with inconsistencies in the language such as context-dependent operators, differences between functions and operators, operators which can be usefully extended to arrays and the branch operator which violates several of the design criteria mentioned by Iverson.

By applying the design criteria more rigorously the power and generality of the language may be increased. Chapter III attempts to do this. A new data type along with changes to the syntax and definition of the language is shown to eliminate some of the problems described above. A new operator - Rename - is proposed to make the additions to

the language more widely applicable. An I/O facility for sequential files is included and made to resemble the present use of variables. Iverson's suggestion that the sequence of operations should be clear is used as justification for eliminating the branch operator and replacing it with the syntax IF, THEN, ELSE and REPEAT, END. The motivations for the changes in Chapter III are increased consistency and the development of a more powerful and general programming language.

One of the concessions made to ease the task of implementation was the altering of the syntax of Iverson's APL so that the language became essentially a language of operators. In Chapter IV a proposal for an implementation by Abrams is examined. The proposal takes advantage of the operator nature of the language. Abrams[1] shows that some expressions in the language can be simplified so that fewer operations need to be done to execute an expression. In addition all expressions in the language can be executed more rapidly if the execution is organized so that slow-speed memory is referenced as seldom as possible. Abrams' proposal is described in terms of a machine with three parts. The first part is a pseudo-compiler, the second an algebraic simplifier, and the third an executor. The pseudo-compiler generates code for the algebraic simplifier

from APL text. The algebraic simplifier interprets its code to produce simplified code for the executor. The simplifier uses the stack-oriented nature of Abrams' machine to generate executor code that will reference slow-speed memory efficiently.

The modifications proposed to the language attempt to retain the simplicity and consistency that APL is known for while introducing facilities to make it a more powerful and useful language. Abrams' machine and the modifications proposed to it form the base for either a software implementation or a hardware design.

CHAPTER II

Criticism of APL

APL is usually described as an array-oriented language, although it could just as well be characterized as a language of operators. These two aspects are fundamental to the formation of APL expressions. The language is designed to achieve the following goals:

1. To have a simple and explicit syntax for all the primitive operators in the language.
2. To extend in an obvious manner the definition of primitive operators to arrays or other data types incorporated in the language.
3. To be as consistent as possible without reducing the power of the language. The consistency should be such that it enhances the simplicity of the language. For example, operands should have consistent conformability requirements, functions should closely parallel operators in their usage, and added features such as new data types or improved I/O facility should be included with as little change to the framework of the language as possible.

In the remainder of this Chapter the discussion will centre

on the APL language - the formation of expressions - as distinct from system commands, or function definition facilities included in implementations.

A variable, a number, a niladic function, and a niladic operator are the simplest APL expressions. More complex expressions can be formed from monadic and dyadic functions and operators with expressions as their operands. Functions differ from operators in that they are not predefined in the language. This distinction is important in the design of an implementation but has relatively little effect in the language. Expressions are interpreted right to left unless parentheses are used to delimit complex expressions as left operands. These few simple rules form the basis for constructing APL statements. Reduction and inner and outer product are useful features in the language which provide a flexible means for forming many new monadic and dyadic operators. Although APL is noted for its consistency[4] there are a few areas which are inconsistent. Some of these areas, such as subscripting, functions, specification, branching and the definition of mixed operators, are covered in the remainder of this Chapter.

2.1 Subscripting

Subscripting, although classified as a mixed operator, does not generally allow as an operand an expression that is valid in any other context except subscripting. It also may take precedence over other operators, a violation of the right to left parsing procedure. These discrepancies are a result of defining subscripting in a manner similar to the classical form wherein the subscriptee is immediately followed by a delimiting pair of brackets which contain the indices. The indices are separated by delimiters such as semi-colons.

APL has generalized the classical definition so that the subscriptee may be a variable, a niladic function that returns a result, or an expression. The individual indices must be valid expressions. The number of indices must be equal to the rank of the subscripted expression. The rank of the result is the catenation of the ranks of the indices. Thus a vector of length one can be subscripted by an n-rank array to produce an n-rank array.

Despite the generalization of the definition of subscripting the inconsistencies incorporated reduce the utility of the language. For example, Index is the only operator which reflects the rank of the operand in the

syntax. This inconsistency leads to the following difficulty. If a dyadic function is to be written which has as a left operand an array of variable rank and dimensions and as a right operand a vector which specifies a scalar component of the array to be returned as a result, the function can be defined as:

```
     ∇ Z←ARR SINDEX V
[1]    →1+⍴V,Z←⍳0
[2]    →0,0⍴Z←ARR[V]
[3]    →0,0⍴Z←ARR[V[1];V[2]]
[4]    →0,0⍴Z←ARR[V[1];V[2];V[3]]
         .
         .
         .
[N]    →0,0⍴Z←ARR[V[1]; ... ;V[N]]
     ∇
```

Examination of SINDEX shows clearly the awkwardness that results from the definition of subscripting. The definition of SINDEX could be extended by allowing an array as the right operand.

However the definition of SINDEX cannot be extended so that it could be used in place of the operator Index. The problems inherent in extending SINDEX are demonstrated by the following examples:

1. $A \leftarrow B[I]$
2. $A \leftarrow C[I;J;K]$
3. $A[I] \leftarrow C$

In the first case the indexing operation can easily be replaced by a function. However the second case demonstrates that subscripting accepts as operands expressions which are valid only in the context of subscripting. The third case demonstrates that subscripting is really two different operators - if it is an operator - which is dependent on the syntactic context for the interpretation of its meaning. It can be an operator which fetches values from an array or in conjunction with Specification stores values into an array. The second and third examples demonstrate that subscripting cannot in general be replaced by a function. The result is, that when proposals are made for alternative definitions of subscripting[7], new primitive operators have to be added to the language.

The design of the language should not be such that it forces a user to implement primitive operators in order to reasonably solve his specific problem. Rather, primitive operators should be incorporated in the language because of heavy use or because they introduce a facility that otherwise could not be obtained.

2.2 Functions

Functions allow the power of the primitive operators to be extended[4]. Functions may be classified in six ways. They are niladic, monadic or dyadic depending on the number of operands and they may or may not return an array as a result. A niladic function that returns a result can be included in an expression as if it were a variable. Monadic and dyadic functions that return results can be considered as operators in most cases except that present implementations do not allow dyadic functions to be used in reduction, or inner or outer products.

There are two methods by which arguments may be passed to an invoked function. Two, or fewer, arguments may be passed as explicit operands to the invoked function, and a result may or may not be returned. The values of the explicit operands will not be changed on return from the invoked function. This parallels the definition of scalar operators. Arguments may also be passed implicitly. Any variable which is defined at the point of function invocation will be known within that function by the same name unless it is declared local to that function or has the same name as one of the explicit operands in the function header. In order to pass three or more distinct arguments to a function there must be a deliberate correspondence

between variable names in the called and calling function. Inadvertent correspondences can, on occasion, lead to rather subtle interaction between programs.

Functions and operators differ in the way in which they pass arguments. A function's arguments are passed by value while an operator's arguments are passed by name[3]. In APL the facility for passing arguments by name is different than ALGOL's facility as expressions are evaluated before operators are invoked and thus expressions cannot be passed as arguments. The essential difference between call by name and call by value is that call by name allows access to the arguments of a routine as known at the point of invocation whereas call be value has duplicated the arguments and these new arguments are kncwn only within the invoked routine. As a consequence the Specification operator cannot be replaced by a function. Thus, it is open to the same criticism as was subscripting. Another aspect of passing arguments by value is that expressions can be formed which are valid and executable, but because of implementation restraints cannot be replaced by a function. For example:

1. $A \leftarrow T[\Delta T \leftarrow \rho B]$
2. $A \leftarrow FUNC\ B$

The array B can be made sufficiently large so that example one will execute and example two will not. The problem in

example two is that the array B needs to be allocated twice in memory so that it can be passed by value. Although this is an implementation restriction it is artifically created by the design of the language. The beneficial aspect of call by value is that it need not be a matter of concern for novice programmers. An implementation designed to serve this group would do well to retain this feature. In a more general environment the facility and efficiency of call by name is worth having.

2.3 Branching

The definition of a function consists of two parts, the function header and the body. The body consists of numbered expressions. In order to pass control from one expression to any other expression a branch operator is available. Branch is a specialized monadic mixed operator. The operator is context-dependent in that it must be the left most character in the string defining a given line in a function. It is the only primitive operator that does not define a result. In larger programs careless use of the Branch operator can eliminate any correspondence between flow of execution and physical layout of the program. The result can be a program that is extremely unreadable and difficult to debug.

2.4 Mixed Operators

Two of the goals for APL mentioned at the beginning of this Chapter were to provide a simple syntax for operators and an obvious extension of the syntax for arrays. The definition of the scalar operators achieves this goal quite adequately. The definition of mixed operators and functions falls short when we consider the constraint of keeping the language as consistent as possible. However consistency of the language must yield to the power and facility provided by the mixed operators. Mixed operators are distinguished from scalar operators because of the variety of restrictions they place on their operands. For instance, the definition of compression or expansion requires a vector or a scalar as the left operand. This precludes conformability requirements associated with scalar arithmetic operators. Some of the mixed operators can be extended so as to reduce the restrictions on their operands and provide a more powerful facility.

Encode and Decode can easily be extended to arrays. This extension exists in some implementations. Any two arrays are conformable for Encode. The resulting dimension is the catenation of the dimensions of the operands. The original definition of Encode allows a vector as the left operand and a scalar as the right. This can be extended so

that column vectors from the left operand are applied to the scalar elements of the right operand array. If the left operand of Encode is an array then Decode can not be used as an inverse function. Decode is extended to accept only a vector or scalar as the left operand and requires the left operand to be conformable to the first dimension of the right operand.

Index Generator, monadic IOTA, accepts as an operand a scalar or one element array. The resulting vector generates indices for other vectors or by repeated use will generate indices for arrays of higher rank. Like Encode and Decode there is a natural extension to Index Generator which allows it to generate the indices for an n-dimensional array. The operand is a scalar or vector which has values that set the upper bound on values in the rows of the result. If the operand is a vector V then the result is a $((\rho V),\times/V)$ matrix. The columns of the matrix are the indices of one scalar element in the array. The extended operator would be equivalent to the following function:

```
∇ Z←INDEXGEN V
[1]   Z←((ρV),×/V)ρIORG+V⍉(⍳×/V)-IORG
∇
```

```
      INDEXGEN 5
1 2 3 4 5
      INDEXGEN 1 2 3
1 1 1 1 1 1
1 1 1 2 2 2
1 2 3 1 2 3
```

Note that the Index Generator used in the definition of INDEXGEN is intended to be the original rather than the extended definition. Also IORG represents the value of the current origin. The result of INDEXGEN is a vector or a rank two matrix. The rank two matrix cannot be used directly with the Index operator to obtain the desired result but can be of use with an INDEX function defined as:

```
      ∇ Z←ARR INDEX IND
[1]   Z←(,ARR)[IORG+(⍴ARR)⊥IND-IORG]
      ∇
```

This is one of the simplest of numerous useful definitions for INDEX. Note that INDEX is an example of a function and not a suggestion for an extension of a primitive operator.

Index Of, like Index Generator, has a restrictive definition. The left operand must be a vector. INDEXOF, shown below, is an extension of the operator Index Of that is consistent with Index Generator.

```
    ∇ Z←A INDEXOF B
[1]   Z←((⍴⍴A),⍴B)⍴IORG+(⍴A)⊤,((,A)⍳B)-IORG
    ∇

    ∇ Z←A INDEXOF B;T
[1]   Z←(2,⍴A)⊤,((,A)⍳B)-IORG
[2]   T←⍉((+/Z[1;]),1+⍴⍴A)⍴1,⍴A
[3]   Z[;Z[1;]/⍳×/⍴B]←T
[4]   Z←((⍴⍴A),⍴B)⍴IORG+1 0↓Z
    ∇
```

```
      A←2 2⍴⍳4
      B←3 3⍴⍳4
      A⍳B
1 1 2
2 2 3
3 3 3

1 2 1
2 3 3
3 3 3
```

The Index Cf operator used above is the restricted and not the extended definition. Two definitions are given for INDEXOF. The first illustrates the relevant points of the algorithm. The additional logic of the second is only for the purpose of inserting (1+⍴A) for indices corresponding to

elements in B which are not in A. The result can be interpreted as follows. If R is the result of A IOTA B and A and B are rank two matrices then the value of B[BR;BP] can be found in A with indices R[;BR;BP]. For example the value 4 is in B at B[2;1]. R[;2;1] is the vector 2 2. Therefore A[2;2] has value 4. The result of this extended Index Of can be used in the function Index above. The extensions of Index Of and Index Generator are therefore consistent.

The criticisms of this chapter point out some of the areas of inconsistency in the current APL. The Subscripting and Branching operators demonstrate weaknesses in the syntax while functions, Encode, Decode, and the mixed operators are examples of weaknesses in the semantics. For the most part the criticisms indicate the difficulties encountered in implementing a language that was not specifically designed to be implemented.

CHAPTER III

Improvements To The Language

The improvements proposed in this Chapter are intended to prcvide partial answers to the adverse criticism of the previous Chapter. The inclusion of a new data type, the mixed-data type, and a new operator, Rename, would solve some of the difficulties associated with indexing and functions mentioned previously. The new data type has the additional benefit of allowing some algorithms to be easily implemented that could only be done in an awkward and inefficient manner previously. Scme of the current implementations have included an I/O facility through extended definitien of operators and functions. An alternate approach that associates symbolic names with files is suggested as a more consistent extensicn to the language. The branch operator is replaced by the syntactic ccnstructs of IF, THEN, ELSE and an iteration clause REPEAT, END.

3.1 The Mixed-Data Type

The proposed mixed-data type is a scalar or array in which each element is a scalar or array of any dimension and attributes except that cf mixed-data type. From the pcint of view of the implementation a mixed-data type vector is

similar to a parameter list in other languages. Thus the values of a mixed-data vector would be pointers to other arrays or scalars. A new dyadic operator for creating mixed-data type arrays is ";". The following examples illustrate its use:

1. *A←I1;I2;I3*
2. *B RENAME I1;I2;I3*

The second example will be referred to in the following section discussing Rename. It can be seen in the first example that the operator ";" is derived from the present syntax for indexing. It is also used in a rudimentary form to output values of dissimilar data types. In statement one, A becomes a three element vector whose scalar elements are the scalars or arrays I1, I2 and I3. For example, I1, I2 and I3 might be, respectively, a character array, a scalar real number, and an empty vector. Other such diverse combinations are possible. The scalar components of A can subsequently be accessed by indexing A. However the rank and dimensions of A[2] are not necessarily that of a scalar. In the example above A[2] would have the same attributes and dimensions as, and could be used interchangeably for, I2 until a Specification operation changed the values of A[2] or I2. If I2 is a rank three array then (A[2])[1;2;3] would select a scalar element from that array. The definition of the operator ";" parallels the definition of catenation.

The essential difference in effect between Catenate and ";" is the data type and dimensions of the result. The expression A;B may be interpreted as one of three possibilities:

1. Neither A nor B is mixed-data type. Then the result is a two element mixed-data vector in which the two scalar components are identical to A and B even though they may be matrices.
2. Only one of A or B is mixed-data type. If A's type is not mixed-data then the result will be formed by making A a scalar component and "catenating" it to the vector B producing a mixed-data vector. If B is not a vector then a rank error occurs as the "catenation" of a scalar and matrix is undefined. A similar result occurs if B's type is not mixed-data.
3. Both A and B are mixed-data type. The result is mixed-data type and the dimensions of the result and the restrictions on the operands are the same as if the operator had been Catenate.

It is important in the definition of ";" that scalar components of a mixed-data array cannot have the attribute of mixed-data. A data type with this recursive definition could be supported but cannot be handled in a consistent fashion with just ";" because of point (3) above. The monadic use of the operator ";" would be similar to ravel.

If the operand were not mixed-data type then the result would be a one-element mixed-data vector. If the operand were mixed-data type then the result would be a mixed-data vector with the scalar elements taken from the operand in row-major order.

All the primitive operators of APL would be defined for mixed-data operands with the exception of reduction and inner and outer products. Scalar arithmetic operators would add, subtract, etc. the arrays which are defined by the scalar elements of the mixed-data operands. Mixed operators would manipulate just the scalar elements of the operands without altering the arrays to which they refer.

3.2 The Operator Rename

In statement 2 above, B is defined similar to A with the proposed Rename operator. Rename does not reallocate the values as does Specification. Instead the operator allows a value or set of values to be known by two different names. Changes to the values of I1 are reflected in B, thus values associated with B are the same values as those associated with I1, I2 and I3. B differs form A in that the values of A are equivalent but not the same as those of I1, I2 and I3. Mixed-data variables are very similar to parameter lists in other languages. The Rename operator

allows these "parameter lists" to be named and manipulated without unnecessarily having to duplicate the values. The facilities provided by Rename do not exist in any form in the language. Thus Rename would have to be added as a new primitive operator.

3.3 Functions

Function invocation would be changed so that operands are passed by name rather than by value. This change will have several effects. Programmers will be able to replace operators such as Specification and Index with functions of their own definition. Implementations would make better use of memory as they would not allocate and move arrays needlessly. Programs will not be recursive by default. Recursion can be forced as follows:

```
1.      ∇ Z←FAC N
    [1]   →0×⍳Z←N=0
    [2]   N←N-1
    [3]   Z←N×FAC N
        ∇
```

```
2.      ∇ Z←FAC N;T
    [1]   →0×⍳Z←N=0
    [2]   T←N-1
    [3]   Z←N×FAC T
        ∇
```

Both examples are simple functions to calculate the factorial function. However the first example will evaluate to zero for all positive values of N. The error occurs because N is being passed by name and its value is being

decremented to zero. Thus N x FAC N evaluates to zero. The second example uses the local variable T to explicitly force recursion.

3.4 Subscripting

The operator Index is considered in its two modes, retrieving values from, and assigning values to arrays. The retrieval mode A[2;3], would have the paired brackets replaced by a single graphic in place of the left bracket. The result would remain the same only the syntax is changed. For assigning values the Index operator is eliminated. Its function is taken over by Specification. The syntax is:

1. *(ARR;I1;I2;...;IN)←B*
2. *ARR←B*

In both statements the left operand of Specification is a valid data type represented by a valid expression. The Specification operator need no longer be context dependent to distinguish between the two statements as the left operands are different data types. In statement 1. the array being indexed is the first element of the left operand and the indices are all the succeeding elements. If any of the indices I1 through IN are empty vectors(⍳0) then ARR will have values specified throughout the range of these indices. This is the equivalent of the expression

ARR[*I*1;;;]←*B* in the present implementation. These changes to subscripting have two advantages:

1. Index is no longer context dependent.
2. With the changes to function invocation, functions can be written which subscript arrays in a different manner than Index[7]. Without these changes the only method of incorporating a different subscripting facility is to add a new operator to the implementation.

The disadvantages of these changes are:

1. The Index operator does not stand out as clearly as do bracketed expressions. Thus complex expressions become more difficult for people to analyze.
2. The language becomes slightly more verbose as an empty vector must be explicitly specified to index through an entire dimension.

3.5 Context-Dependency of Expressions

In current implementations there are two types of statements - specification and branch statements. Neither of these causes any I/O to be done implicitly. An expression which cannot be categorized as either specification or branch is arbitrarily categorized as a specification statement with as implicit Quad and

Specification to the left of the expression. Consider the following expressions:

1. $A \leftarrow 5$
2. $(A \leftarrow 5)$

Statement 1. causes no I/O whereas statement 2. does. This context dependency could prove inconvenient in situations where specification or subscripting has been replaced by a function. For example, if a function SPEC has been defined similar to the operator Specification, then a statement consisting of the expression A SPEC B would cause the values of A to be output. This output should not be forced on the programmer. Rather, context dependency of expressions should be eliminated and the programmer must explicitly indicate at what point he wishes I/O to be done.

3.6 I/O Facility

I/O facilities are being added to some APL implementations[9] by extending the definition of some operators such as Ibeam. These operators are then imbedded in locked functions to make their use easier. Although operators are part of the language the result of this ad hoc approach is that I/O is being added to the implementation and not the language. Consequently there is little likelihood of compatibility of programs between

implementations.

If we analyze the present I/O facility of APL we find that Quad has properties of both variables and niladic functions. Quad cannot be a niladic function because a function cannot have values specified to it as in the following expression:

```
⎕←A
```

If we consider Quad to be a symbolic designation of a special variable - despite the inconsistencies - we can expand this primitive facility into a more general form. This can be done by adding a mechanism to the language which allows any valid symbolic name to be declared as an I/O variable. There exists in the language a mechanism for declaring - defining - symbols to be functions. A similar mechanism would declare a symbol to be an I/O variable. A symbol could then represent one of three things, a variable, an I/O variable, or a function. From the point of view of the implementation an I/O variable would be quite distinct from a variable. However, in the language they should be as similar and interchangeable as possible.

When defining I/O variables one of two types, a simple type and a complex type, would be specified. This information is primarily for the implementation's I/O support routines but would have effects in the language. On

output through a ccmplex type I/O variable both the data and its descriptor would be written to the I/C device. Details concerned with blccking, the host operating system, the machine model, and the format the data is written in would be specific to the implementation. Thus data in this format is likely to be incompatible with other implementations and I/O devices. On input the data is self-describing and the I/O variable virtually indistinguishable from a variable. The complex type is intended for data bound for intermediate mass storage. Arrays of arbitrary rank and attributes can be output and retrieved without recourse to formatting schemes and intermediate data forms. As the descriptors are included in each I/O operation there is nc necessity for conformability between two successive I/O operations.

The simple I/C variable is a more powerful and general facility but very inflexible in its use. On input or output the I/O variable must be a boolean vector. Cn input a block of data received from an I/O device will be considered as a string of bits which will determine the values of a boolean vector. The length of the string will be determined by the number of bits received from the device. Any communication control characters in the string would not be stripped by the I/C support routines. Output must be a boolean vector which has any necessary control characters included in the

string. The primitive I/O facility is designed to get the data in and out of memory in a form characteristic of most digital computers. Formatting functions written in APL can provide the transformation of data into a convenient form.

The appearance of I/O variables in the language is similar to variables. For example:

1. *A←FILEIN*
2. *FILEOUT←B*

Assume A and B are variables and FILEIN and FILEOUT are I/O variables. If FILEIN is simple then A will be specified as a boolean vector. Otherwise the descriptor of A will be determined by the descriptor accompanying the data which can be assumed valid. If FILEOUT is simple then B must be a boolean vector. If FILEOUT is not simple then the descriptor of B will be preserved. Each time an I/O variable appears as an operand in an expression being executed an I/O operation is performed to transfer the appropriate values.

I/O variables differ from variables in that they can not be subscripted when they are the left operand of Specification. The restriction arises in the following expression:

(FILE;I1;I2)←A

If, in the process of declaring FILE, it was declared

"write-only" then an error will be indicated. The indices are not used to access an "indexed" file as this is not the interpretation that would be applied if the file were an input file. If FILE is "read-only" then a different error will be indicated.

3.7 IF, THEN, ELSE and REPEAT

The control structures IF, THEN, ELSE and REPEAT, END are to be included in the language and the Branch operator eliminated. There are two reasons for eliminating Branch. Primarily, the alternative control structures make many programs easier to understand and debug[16], and secondly, the syntax of the Branch operator is not consistent with the rest of the language. Eliminating Branch also eliminates the possibility of constructing iterative groups of statements. REPEAT, END replaces this facility with a syntax that more explicitly defines the group. The syntax for these control structures is:

1. IF (BOOLEANEXP) THEN (EXPS1) ELSE (EXPS2) END
2. REPEAT (REPEXP)
   (EXP1)
   (EXP2)
   (EXPN)
   END:LABEL

The symbols that are not parenthesized are constant and must appear as shown for the syntax to be correct. Parentheses and parenthesized symbols are to be replaced by an expression or expressions. The results of example 1. are well known from AIGOL. BOOLEANEXP represents a single expression which evaluates to a scalar or single element array which is boolean. EXPS1 and EXPS2 represent zero or more expressions. A null expression results in no execution. In example 2., REPEXP is a single expression which evaluates to a scalar or single element array that is a positive integer. It specifies the number of times the group of expressions; EXP1, EXP2, EXPN, will be executed. END or END:LABEL delimits the group. LABEL is a string of characters that uniquely identifies, within a function, the END of a REPEAT group. The rules for forming strings which are valid labels are the rules for forming variable names. Two control structures associated with REPEAT are CYCLE and LEAVE or CYCLE:LABEL and LEAVE:LABEL. CYCLE causes execution of the next iteration to begin immediately with the first expression in the inner-most REPEAT group in which CYCLE appears. LEAVE causes execution to continue with the first expression after the inner-most REPEAT group. If a label is suffixed to the CYCLE or LEAVE then the REPEAT group that execution is transferred to is the one identified by the label. This facility is useful when execution is to

be transferred out of several nested REPEAT groups[18]. Errors would be indicated if the label were not defined within the function or if the CYCLE or LEAVE was not within the REPEAT group that the label was attached to. An example of a bubble-sort program illustrates the use of this new syntax:

```
        ∇ SORT←GRADEUP V;D;I;J
    [1]   SORT←⍳I←⍴V
    [2]   REPEAT I←I-1
    [3]   FLAG←0×J←⍴V
    [4]   REPEAT I
    [5]   IF V[SORT[J-1]]>V[SORT[J]]
    [6]      THEN D←SORT[J-1]
    [7]           SORT[J-1]←SORT[J]
    [8]           SORT[J]←D
    [9]           FLAG←1
    [10]     ELSE END
    [11]  J←J-1
    [12]  END
    [13]  I←I-1
    [14]  IF FLAG THEN CYCLE ELSE LEAVE END END
        ∇
```

There are several implicit points in the program above which should be stated. One is that repeat groups are independent of subsequent values of variables that initialize them.

Thus when line 2 is executed a counter is set up which is not affected by changes in the value of I. The second point is the method by which expressions are delimited. When there are several expressions to be executed the only method of delimiting them is to list them on separate lines as shown by lines 6 through 8. The third point is the inconsistency of the reserved words added to the language. This objection could be overcome by replacing the reserved words with single character graphics. There would then be considerable similarity between the new control structures and parentheses. However, a considerable degree of intuitive appeal is lost by the replacement of well known symbolic names. As the reserved words have well known meanings it is suggested that they be retained. The fourth point is the need for preprocessing of programs. Consider the following statement:

IF DIR=0 *THEN I*←1 *ELSE I*←*ILAST END*

This statement cannot be executed right to left as other statements can. Thus a preprocessor must break it up into three separate parts so it can be properly executed. In current implementations this would be awkward. However, if the implementation is done as suggested in the following Chapter this preprocessing could easily be incorporated in the pseudo-compiler stage.

Any program that will terminate in finite time can be represented by three basic control structures[16]:

1. Simple sequencing
2. IF, THEN, ELSE
3. REPEAT, END

Programs written with these control structures are characterized by one entry at the start of the program and one exit at the end. The control structures ensure that execution takes a direct path from entry to exit. Thus all programs written will be very similar to the straight-line algorithms or method of leading decisions that Iverson advocated in his book "A Programming Language." In addition all non-recursive programs must terminate in a finite time. All loops must use REPEAT which has a finite replication factor. These factors combine to make programs easier to understand and debug.

3.8 Survey of Other Improvements

A. L. Anger[2] has suggested the addition of an embedded dyadic branch operator. This would allow expressions of the following form:

$$Z \leftarrow A + J \rightarrow LABEL$$

In this expression if LABEL were not iota zero then the branch would be taken and the left operand of the branch

operator would be left unevaluated. If IABEL is iota zero then the result of the branch operator is simply the left operand J. Although this suggestion eliminates the syntactic weakness of the branch operator it introduces the ability to construct APL statements that are very obscure.

J. Ryan[17] has suggested the introduction of lists as a data type to the language. A ";" operator would be used to construct the lists. The following expression would generate a binary tree, T, with leaves A, B, C, and D:

```
T←((A;B);(C;D))
```

The data structures that can be created using this operator are superior to the mixed-data type mentioned above. However the operator has been made context dependent in order to achieve these structures. The removal of superfluous parentheses from the above expression changes the shape of the tree obtained.

S. Charmonman[11] has proposed a modification to the definition of conformability for the dyadic scalar arithmetic operators. Charmonman proposed an extended definition that would allow any two operands to be conformable. In the case that the ranks of the operands are not the same then the operand with the smaller rank is reshaped to the rank and shape of the operand with the larger rank. If the ranks are identical but the operands

have different dimensions then the operand with the least number of elements is reshaped to the rank and shape of the operand with the greatest number of elements. If the dimensions are different but the number of elements in the operands is the same, then the left operand is reshaped to match the right operand. However, as L. Breed[10] has pointed out, this generalization to the definition of conformability destroys the properties of associativity and communtativity.

The APL PLUS file subsystem was designed by L. Breed[9] and E. B. Iverson to add I/O capability to APL. Their design objectives were:

1. Operation with APL/360 with no changes to the language.
2. Efficient operation with arrays as objects of data transfer.
3. File access does not envolve extralingual system commands.
4. File sharing and high reliability.

To achieve these objectives a single primitive operator was introduced. Then, for user's convenience, functions FTIE, FCREATE, FREAD, FAPPEND, FREPLACE, FRDCI and FHOLD were written all using the new primitive operator. FTIE and FCREATE associate a file number with the file's name.

FCREATE in addition creates a new file. FREAD retrieves an indexed component of a file. FAPPEND adds a component at the end of a file and FREPLACE replaces a component. FRDCI returns component information about a file such as the account number the component was written under and the time at which it was written. The FHOLD function allows synchronization between two people updating a file simultaneously.

Compared to other languages it can be seen that there are relatively few and simple concepts to be mastered in order to be able to obtain a sophisticated I/O capability. In addition it is relatively simple to add this capability to current implementations. However, by adding the I/O capability to the implementation rather than the language, consistency between implementations may well be lost.

CHAPTER IV

Improvements To The Implementation

Current implementations of APL are simple interpreters. Character strings used to define programs have simple editing operations, such as removal of excess blanks and some standardizing of format, performed on them before they are stored. However, a high degree of visual fidelity is maintained. The interpreter uses the edited strings for execution of programs. It begins its analysis of the strings on the left to check for possible comment statements or system commands and then continues with a right-to-left parse and execution of the string. The visual fidelity retained in the strings[8] used to execute programs results in extra processing in the identification and resolution of operands, the identification of operators and the detection of simple syntactic errors. Preprocessing of text can make operator identification and operand resolution significantly more efficient in terms of processing required. Considerable efficiency in memory space and execution speed can be gained by deferral of operators and simplification of expressions as proposed by Abrams[1]. Deferral of operators results in reduced memory requirements for temporary results and less computation for memory management associated with

temporary allocations. The following expression illustrates an ideal situation for deferral of operators:

$A \leftarrow B1 + B2 + \ldots + B9 + B10$

Current implementations allocate storage for a temporary result of B9 + B10 and then proceed with the addition. Allocation of a temporary result is not needed in this example. By deferring the add operation until the Specification operator is interpreted all the addresses of operands of add can be resolved. If the interpreter then adds only one scalar component from each operand and stores the result in A, temporary storage for only one scalar is needed. The process is repeated for the next scalar component of the operands. The scalars are accessed in row-major order. After $(\times/\rho B10)$ repetitions A will be completely specified. As only one element of temporary storage is needed, a register can be allocated for this purpose with significant savings in processing time. Savings in execution time derived from the simplification of expressions is illustrated by the following expression:

$A \leftarrow (1\ 1\ 1) \circ . \times B + C$

The result A, is just the diagonal sum of the arrays B and C. Current implementations in producing the temporary sum of B + C do much unnecessary processing in summing the off-diagonal components. Abrams proposed a simplification scheme, called beating, so that the expression above would

be calculated as if it were:

$A\leftarrow((1\ 1\ 1)\otimes B)+(1\ 1\ 1)\otimes C$

Beating is the process of determining equivalent expressions which take fewer operations to calculate than the original expressions. Abrams showed that a subset of APL operators produced expressions that are beatable. The above example would require processing time that is expcnentially related to the number of scalar components of the arrays for current implementations whereas Abrams' proposal takes processing time that is linearly related.

Abrams stated his proposal as a design for a machine - the APL machine or APLM. Although it is not necessary to consider it in machine-like terms, Abrams' proposal will be referred in terms ascribed to machines in the following Chapter. Many of these terms have equivalents in a software implementation. Fcr instance, registers rather than arrays, and bits rather than flags will be described.

A software implementation of Abrams' APLM has been undertaken by the author and D. A. James. James worked on the DM and the author on a subset of the EM. The EM was coded in IBM/360 Assembler Language. The implemented subset demonstrated that the fundamental design of the EM is correct by properly executing expressions that confined their operations tc those of the subset. In addition minor

discrepancies in the logic as proposed by Abrams were found.

4.1 The APLM

The APLM consists of three submachines, the C, D, and E machines. The purposes of these machines will first be described briefly, and in detail later. The C machine (CM) "compiles" raw APL text into an efficient form for execution by the D machine (DM). The DM analyzes this code into beatable and deferrable expressions. When an operator is encountered that cannot be beaten or deferred the DM produces code for the E machine (EM). The EM is given control to execute the generated code and produce the result. It is reasonable, although redundant, to have three versions of a program as the different versions are expected to reside in different storage hierarchies. EM code may be imbedded in several nested loops and therefore resides in high-speed register storage. DM code is needed once for every invocation of the program in which it belongs and therefore resides in intermediate core storage. CM code is only needed once to generate DM code and possibly for the generation of error messages and can therefore be placed on a mass storage device.

The CM does not truly compile APL text as it does not resolve the addresses of operands and only modifies

operators to make their identification a rapid process. It does compile in the sense that it reduces text to an opcode, operand form similar to one and two-address machines. The DM handles some operators such as monadic RHO by itself, but it is essentially a run-time "algebraic simplifier". These simplification processes cannot be done by the CM as they require the values of variables that are not determined until run-time. The CM is included as part of the machine, rather than simply being part of the software, so that the machine can be used recursively to evaluate an operator such as Unquote.

4.1.1 Registers in the APLM

The APLM consists of stack and scalar registers, core memory, and the processing units for the C, D, and E machines. The C, D, and E machines all have access to the registers and core memory. The registers are as follows:

1. Iteration Stack (IS)

CTR	MAX	DIR	CH	MRK	
0	5	0	0	1	
0	9	0	0	0	
3	3	1	1	0	<= Top Entry

The IS is effectively a set of nested loops. The ISCTR is the CTR field of the entry at the top of the stack. It

should support values as large as the largest integer to be supported by the implementation. It should be able to take on values at least as large as the number of bits in core in order to effectively index large boolean arrays. ISCTR is incremented or decremented through the range of values associated with the loop. ISMAX specifies the initial or final value to be taken by ISCTR. ISDIR is a boolean value. If ISDIR is zero then ISCTR is initialized to zero and incremented to ISMAX, otherwise ISCTR is initialized to ISMAX and decremented to zero. If the above set of loops were being used to access the scalar components of a rank 3 array then the dimensions of the array as derived from the ISMAX fields are 6, 10, 4. The ISMAX fields are one less than the dimensions as ISCTR is used to calculate a displacement from the origin of the array. ISCH and ISMRK are boolean values. ISCH is used to reduce the number of calculations required to determine the value of the polynomial access function. The details of its function are explained in the section on the EM. ISMRK is used to delimit groupings within the IS. These groupings may correspond to the dimensions of arrays, reduction operators, or REPEAT groups. Its function will be more clearly explained under array accessing in the EM section.

2. Location Counter Stack (LS)

REL	ORG	LEN	CDE	IS	FN	NWT	QP	
10	200	50	00	0	0	0	0	
0	0	10	01	1	0	1	0	<= Top Entry

The IS functions as a location counter that makes DM and EM code readily relocatable. A segment of DM or EM code is invoked by pushing a descriptive entry to the LS. LSCDE specifies which of the three machines to be given control. A logic unit, called Maincycle, is given control after every DM or EM instruction executed. Maincycle uses LSCDE to select the appropriate submachine to receive control. Depending on which machine has control the application of LSREL, LSORG and ISLEN is modified. In the DM LSORG specifies the starting location of a segment of code in memory, ISLEN delimits the extent of the segment, and LSREL selects the instruction, relative to LSORG, to be executed. In the EM the application of these fields are similar but apply to the Instruction Stack (QS) rather than to core storage. LSORG should be capable of specifying any core address. LSREL and LSLEN need not contain values as large as LSORG but smaller values artificially constrain the maximum size of programs. LSIS, LSFN, and LSNWT are boolean values that, along with LSQP, preserve information associated with the program segment. LSIS has a value of

one if the program segment is being repeated under the control of the IS and zero if not. When ISREL equals LSLEN the LS entry is popped unless LSIS is one. If LSIS is one, LSREL is set to zero, the IS is incremented - stepped - and the segment repeated unless the IS has completed - overflowed - its range of iterations. If the IS overflows then the top LS entry is popped regardless of LSIS. LSFN is one if the program segment the LS entry defines is an APL function and zero otherwise. LSNWT saves the value of the scalar register, NEWIT, when another LS entry is pushed to the stack. LSQP is a pointer into the QS. It is used in the evaluation of subscripted expressions. Its function is further explained in the EM section under subscripting.

3. Instruction Stack (QS)

```
CP    ADDR
S      5
JMP    10    <= Top Entry
```

The EM places the instructions it creates for the EM in the QS. The QS serves as a buffer for the EM instructions and as a stack for data associated with the EM's operation. Although some of the operations performed on the QS are stack operations many are not. Thus its designation as the Instruction Stack is a misnomer. The instructions shown are illustrative of EM instructions. These instructions are

discussed further in the section on the EM.

4. Value Stack (VS)

TAG	VALUE	
RT	0	
ST	5	<= Top Entry

The VS serves a variety of purposes. The DM uses it to temporarily store program segment descriptors, addresses, names, etc. The EM uses it to store temporary results during the evaluation of expressions. This is the main stack in the machine and it performs a wide variety of functions associated with many parts of the machine. Its basic purpose is the storage of temporary results. VSTAG specifies an interpretation to be associated with VSVALUE. The tag ST identifies 5 as a scalar value.

4.1.2 Core Memory in the APLM

1. Memory

Memory contains arrays, array descriptors, a free pool area, and freed spaces. Arrays are kept in the low end of core with freed arrays in a doubly linked list. Array descriptors are kept in high core and freed descriptors doubly linked. Between the arrays and array descriptors is a free pool area of allocatable storage.

2. Name Index Table (NIT)

The NIT is a N,3 matrix. Each row of the matrix represents a symbolic name in the original text. A character vector, that is the concatenation of all the unique symbolic names encountered in the text, is maintained in association with the NIT. The first column of a row is the index in the character vector of the first character in the name. The second column is the length of the name and the third column an INX value to be associated with the name. INX is the value that will be used in place of the symbolic name in the DM code. The rows of the NIT are ordered according to the collating sequence of the names they represent. When a new symbolic name is encountered in the text it is concatenated to the character vector and assigned the next INX value. The INX value is recorded in the row inserted in the NIT.

3. Name Table (NT)

The NT is an associatively addressed memory. Each entry has a name, tag and value field. The name field contains an INX value corresponding to a symbolic name in the original text. The tag field is similar to the tag field discussed in connection with the VS. Tags designate various properties of INX values such as scalar, function, undefined array, etc. The value field is interpreted

differently for each tag. Some value fields contain memory addresses. Others, such as those tagged as scalar, have the value of the scalar in the value field. Abrams included the NT with the registers but its relatively infrequent use suggests it should reside in a lower level of storage.

4.2 The C Machine (CM)

The CM accepts APL text and produces DM code in reverse Polish form. DM code has operators coded as fixed length opcodes which will index the appropriate operator routines when applied to a table of addresses. Operands are also transformed into fixed length codes using the NIT. The resulting DM code has a format similar to the machine code of zero and one-address machines.

4.3 The D Machine (DM)

The DM analyzes expressions in reverse Polish code produced by the CM to produce EM code which will be more efficient than simply executing operators in order of precedence. A subset of the DM instructions with a brief annotation to explain each can be found below. These instructions are represented by an assembly language format rather than numeric values the CM generates. All examples of DM code will use this assembly language.

DM Instruction Set

Load Scalar: IDS scalar

The scalar is an immediate operand. This instruction is used in the expression $K \leftarrow I+1$ to indirectly cause the EM to load the constant 1 to the VS. The DM does this by generating EM instructions and then passing control to the EM for execution.

Load Constant Array: LDCON disp

The operand disp represents a displacement from the beginning of the program that the instruction is in to the location of a descriptor array for the constant. The constants are pooled at the end of the program. In the expression (2 3 5)$\uparrow A$ the LDCON instruction would be used to define the left operand of Take. In the expression (2 3 5)$\uparrow A$ the LDCON instruction would cause the EM to load the values of the constant array.

Load Name and Fetch Value: LDNF name

The operand name is the INX value assigned by the CM. The symbolic name can be recovered from the NIT. By searching the NT for the INX value it can be determined if the name is defined. If the name is defined the NT entry

has the information necessary to access the value. In the expression $K \leftarrow I+1$ the LDNF I instruction would cause the EM to load the value of I.

Load Name: LDN name

IDN is similar to LDNF above. In the expression $K \leftarrow I+1$ the LDN K instruction would cause the EM to load the memory address of the values of K. This address can then be used by the Specification operator to do the assignment.

Load J-vector: LDJ len,org,dir

This instruction is defined similar to the function JFUN defined as:

```
      ∇ Z←JFUN ARG;LEN;ORG;DIR
[1]   LEN←ARG[1]
[2]   ORG←ARG[2]
[3]   DIR←ARG[3]
[4]   IF DIR=0 THEN Z←ORG+(⍳LEN)-IORG
[5]             ELSE Z←(LEN+ORG-1)-((⍳LEN)-IORG) END
      ∇
```

The result of this function is a vector known as a J-vector. J-vectors are an extension of the operator Index Generator. In the expression (1 2 3 4)↑A the LDJ 4,1,0 instruction would define the left operand of TAKE. Notice that the CM will have to detect instances of J-vectors. The efficiency

gained by J-vectors is sufficient to warrant the inclusion of the LDJ instruction.

Assign: ASGN no operand

In the expression $K \leftarrow I+1$ the instruction ASGN would cause the EM to store values in the VS at K. After the values are stored they are popped from the VS.

Assign and Leave Value: ASGNV no operand

In the expression $K \leftarrow I \leftarrow I+1$ the instruction ASGNV would be used for the assignment of values to I. ASGNV is similar to ASGN except that values are not popped from the VS. ASGN would then store the values in K.

Add: ADD no operand

This instruction generates EM code that will add the top two entries of the VS and replace the entries by the result. Add is representative of many other dyadic scalar arithmetic operators. These should be well known from the language and will not be repeated here. They are, however, part of the DM instruction set.

Plus: PLUS no operand

This instruction generates EM code that will perform the monadic plus operation on top of the VS and replace the

entry by the result. Plus is representative of the other monadic scalar arithmetic operators.

Take: TAKE no operand

The right and left operands of TAKE have been pushed to the VS as unevaluated EM program segment descriptors. The right operand is inspected to determine if it represents a deferred beatable program segment. If the segment is not beatable it is evaluated to temporary space. The left operand is then used to beat the descriptor of the right operand. The program descriptor for the left operand is popped from the top of the VS. Take is a selection operator and does not generate any EM code. Take is representative of the dyadic selection operators Drop (DROP) and Transpose (TRANS).

Reverse: REV no operand

The right operand and the co-ordinate the operation is to be applied to are defined by segment descriptors on the top of the VS. If the right operand is not beatable it is evaluated to temporary space. The descriptor for the right operand is then beaten. Like Take, Reverse is a selection operator and generates no EM code. Abrams defined Reverse with an operand K such that the expression $\phi[K]A$ would generate EM code REV K. This is not sufficiently general.

The expression $\phi[+/K]A$ can not be handled as expressions can not be operands of DM instructions. Other instructions that differ in a similar manner from Abrams' are Laminate, Compress, Expand, Rotate, and Reduction.

Decode: BASE no operand

The left and right operands are evaluated to temporary space if they are unevaluated expressions and not variables. The Decode routine is invoked and the result evaluated to temporary space. Decode is representative of a group of operators that are evaluated immediately. These are Encode, Decode, Grade Up, Grade Down, Catenate, Ravel, Dimension, and Restructure.

Index Generator: UIOTA no operand

UIOTA is similar to BASE in that it is evaluated immediately. It is evaluated to a J-vector rather than temporary space. After UIOTA has been "evaluated" the J-vector can be deferred and beaten.

Index: SUBS K

The operand on the top of the VS must be a rank K array. The next K entries in the VS represent the subscripts. The DM generates EM code to fetch the operands.

It should be remembered that DM instructions need interpreting. For example, the instructicns LDNF I and LDN J appear similar to the IBM 360 instruction L 9,NAME. However the latter instruction will have the memory address of NAME resolved when it is executed. The former instructions need interpretation to resolve their memory locations. The operand must be searched for in the NT and found before a value can be accessed.

These instructions can be used to represent an APL program as shown in the following examples:

DM Code

$K \leftarrow J \leftarrow I+1$			$K \leftarrow (J \leftarrow I)+2$		
Address	Operation	Operand	Address	Operation	Operand
500	LDS	1	600	LDS	2
501	LDNF	I	601	LDNF	I
502	ADD		602	LDN	J
503	LDN	J	603	ASGNV	
504	ASGNV		604	ADD	
505	LDN	K	605	LDN	K
506	ASGN		606	ASGN	

Example 1 Example 2

The DM code above is just recoded representations of the

original expressions. Notice that the CM must be sensitive to context in assembling these instructions. In assigning values to J an ASGNV instruction is used. ASGNV creates EM code which will leave values in the VS for further use - in this case for assignment to K. In assigning values to K there is no further need for these values and the ASGN instruction will cause them to be popped. The CM must also be able to detect the double occurrence of J in the following expression:

$$K\leftarrow(\phi J)+J\leftarrow I$$

If the CM uses ASGNV for $J\leftarrow I$ rather than ASGN then erroneous values of (ϕJ) will be used. The need for the CM to be examine the context is due to the design of DM code and not inherent in the source language of APL. Making DM code contextually dependant results in more efficient execution of expressions.

Most DM instructions facilitate either the fetching and storing of variables or operations corresponding to the primitive operators in the language. Instructions which store or fetch variables are used to construct EM instructions which will store and fetch values to and from memory. The DM builds the EM instructions in the QS and pushes a segment descriptor - a location and length description - to the VS. When the DM enccunters an

instruction corresponding to a primitive operator it finds the operands at the top of the VS and checks them for conformability. If the operands are conformable for the operator then the operator is classed as one of immediately evaluated, deferrable or beatable.

An operator is immediately evaluated if it can not be deferred or beaten. The operators that are evaluated immediately are Decode, Encode, Grade Up, Grade Down, Catenate, Ravel, Dimension, and Restructure. Some operators, such as Base, are immediate because they need access to all the values of their operands before any part of the result can be specified. Catenate and Ravel are immediate because of restrictions in the method by which array accessing is done. The accessing method is logically similar to a set of nested loops. Catenate needs two consecutive sets of loops and Ravel needs two parallel loops. Without considerable modification to the APLM these facilities can not be provided.

An expression is deferred by creating the EM code in the QS for later execution. Any expression can be deferred but the benefits of deferral may be negated by the cost of deferring them. Thus expressions are deferred until:

1. The expression is exhausted and must be executed.
2. A function is encountered. Niladic functions are

treated as variables by the CM. The DM must detect niladic functions and cause execution of the expression up to the point where the function was encountered. The niladic function can then be evaluated to temporary space so that the result can be treated as a variable. The function cannot be evaluated sooner as illustrated by the following expression:

```
A←NF+B←L/V
```

If NF represents a niladic function which alters the variables B or V then evaluation of NF before B is specified will likely cause an error. Monadic and dyadic functions must have their operands evaluated before the functions are invoked for similar reasons.

3. Specification or one of the operators that is immediately evaluated is encountered. Specification is not deferred if it has been coded as ASGNV rather than ASGN. This is context dependent.
4. A General Dyadic Form (GDF) operator is encountered. GDFs are a result of inner and outer product operators occurring in the source. In this case the right operand is evaluated to temporary space. If GDF's were deferred then each element of the right operand would be calculated as many times as there

are elements in the left operand. This needless repetition obviously should be avoided.

5. The deferred code exceeds some implementation limits. The size of the QS is likely to be the most restricting limit. Multi-dimensional arrays, even though they may contain only a few elements, require considerable space in the QS relative to scalar operands.
6. The result of an expression is needed for the process of beating. For example:

$$B\leftarrow 5+A\leftarrow(T1+T2+T3)\uparrow M\times N$$

Selection operators such as Take are evaluated by the DM by the process of beating. The left operand of Take can not be used for beating until it is evaluated. The expression $(T1+T2+T3)$ is not deferred. The expression $M\times N$ is however deferred until the DM encounters the specification of B. If the right operand of a selection operator is not beatable then it is not deferred. The method for determining beatable expressions is explained below.

The operators that can be beaten are the dyadic selection operators Take, Drop, and Transpose and the monadic selection operator Reverse. Index can be beaten if the indices of the array are scalars or J-vectors. We would

expect arrays indexed by scalars and J-vectors to be beatable as the same result can be obtained by replacing Index by a combination of Take, Drop, Transpose, and Reverse. For example, if A is a 10 by 10 by 10 array, the following two expressions are equivalent:

```
A[⍳3;,6;4+⍳4]                    (1)

(0,5,4)↓(3,6,8)↑A                (2)
```

The DM beats operators by changing the polynomial access function of the right operand. For example if A is defined as above:

Expression	Polynomial Access Function
`A`	`AV(P,R,C) = 100P + 10R + C`
`(¯2,3,¯5)↑A`	`AV(P,R,C) = 100P + 108R + 8C + 805`
`(2,3,5)↓A`	`AV(P,R,C) = 100P + 10R + C + 235`
`⌽[1]A`	`AV(P,R,C) = -100P + 10R + C + 90`
`⌽[2]A`	`AV(P,R,C) = 100P - 10R + C + 90`
`(1,1,1)⍉A`	`AV(C) = 100C + 10C + C`
	`= 111C`
`(2,1,2)⍉A`	`AV(R,C) = 100C + 10R + C`
`(1,2,1)⍉A`	`AV(R,C) = 100R + 10C + R`

P - Plane R - Row C - Column

P, R, C are zero origin indices

AV <= ,A

The EM does not calculate the polynomial access function in

an obvious fashion. The mechanism used is computationally more efficient and provides better error checking than the evaluation of a polynomial function. As a consequence the beating process by the DM is more involved. All the data involved in beating is contained in the descriptors of arrays. For every variable there is a Name Table (NT) entry which contains the pointer to a descriptor. A descriptor or descriptor array (DA) contains the following information about arrays:

DATYPE - the implementation type, integer, boolean, real, etc., of the values described.

DALEN - the amount of memory allocated to this descriptor.

DAFILL - the amount of memory unused by the descriptor.

DAREFCNT - the number of references to this descriptor. Multiple references may stem from the NT or EM code.

DAVBASE - the memory location where the values are stored.

DAABASE - the displacement from VBASE to the first value. ABASE is generally zero unless beating alters it. Its value is the constant shown in the polynomial access functions above.

DARANK - the rank of the array.

DADIMEN - the dimensions of the array. There are DARANK of these.

DADEL - the coefficients of the polynomial access function. There are DARANK of these. The location of an element of an array is given by *VBASE+(DADEL×INDICES)+ABASE*

In beating an operator the descriptor of the right operand is altered. Beating constitutes the execution of the operator by the DM. It generates no code in the QS for the EM. The algorithms for changing the descriptors are:

1. *Q+A*

   DAABASE←DAABASE+DADEL+.×(Q<0)×DADIMEN-|Q

   DADIMEN←|Q

2. *Q+A*

   DAABASE←DAABASE+DADEL+.×(Q>0)×|Q

   DADIMEN←DADIMEN-|Q

3. *⌽[J]A*

   DAABASE←DAABASE+DADEL[J]×DADIMEN[J]-1

   DEL[J]←-DEL[J]

4. *Q⍉A*

   DARANK←1+⌈/A

   R←DADIMEN

   D←DADEL

   DADEL←DARANK↑DADEL

```
DADIMEN←DARANK↑DADIMEN
I←0
REPEAT (DARANK)
    DADIMEN[I]←⌊/(I=Q)/R
    DADEL[I]←+/(I=Q)/D
    I←I+1
END
```

5. Subscripting by a scalar

   Let K denote that the scalar is the Kth subscript (zero origin)

```
DAABASE←DAABASE+DADEL[K]×SCALAR
DADEL←(K≠⍳DARANK)/DADEL
DADIMEN←(K≠⍳DARANK)/DADIMEN
DARANK←DARANK-1
```

6. Subscripting by a J-vector

   Let K denote that the J-vector is the Kth subscript (zero origin)

   Let the argument for JFUN, mentioned above, be LEN, ORG, S

```
DAABASE←DAABASE+DADEL[K]×ORG+LEN-1
DADIMEN[K]←LEN
IF S=1 THEN DEL[K]←-DEL[K]
```

The mechanism the DM uses to simplify expressions like `Q↑A` is shown above. How this mechanism is extended to

handle expressions in which A is replaced by a complex expression is shown below:

APL Expression: $A \leftarrow (3 \uparrow A1 + A2) - (7 \uparrow B1 \times B2)$

DM Code			EM Code		
Address	Operation	Operand	QSADDR	Operation	Operand
250	LDNF	B2	0	IFA	@B2
251	LDNF	B1	1	IFA	@B1
252	MULT		2	OP	MULT
253	LDS	7	3	IFA	@A2
254	DROP		4	IFA	@A1
255	LDNF	A2	5	OP	ADD
256	LDNF	A1	6	MINUS	
257	ADD		7	OP	ASGN
258	LDS	3	8	POP	
259	TAKE				
260	MINUS				
261	LDN	Z			
262	ASGN				

When the DM encounters the DROP instruction it can determine the operands of the instruction by inspecting the VS. The top entry of the VS indicated a segment of code, a Load Scalar 7 instruction, in the EM. This segment evaluates to the left operand of DROP - which was overwritten subsequently by the IFA @A2 instruction. The entry second

from top of the VS indicated a segment of code, QS addresses 0 through 2, which is the right operand of DROP. In order to beat DROP the DM scanned for IFA instructions in the code for the right operand in the QS. It changed the descriptor corresponding to the operand of the IFA instruction using the values of the left operand as described in the algorithms above. The operand of an IFA instruction, @B2 for instance, is the address of the descriptor for @B2. Having beaten all descriptors it found by means of IFA instructions the DM pops the top entry off the VS and continues processing DM code.

The beating process described above can be applied only if the right operand of the selection operator is beatable. If the operand is not beatable it is evaluated to temporary space and then beaten. The operands that are beatable are:

1. Variables or expressions that the DM has reduced to a J-vector.
2. Expressions formed from a dyadic or monadic scalar arithmetic operator with beatable operands.
3. Reduction with a beatable expression as an operand.
4. The General Dyadic Form. Both operands are always evaluated to temporary space and are therefore beatable.
5. Expressions of the form *NρS1* where S1 is a scalar or

one element array. Restructure is normally evaluated immediately but this special case can be deferred in the QS as:

```
S    S1
IRD  @TEMP,MASK=X
```

The beating process is applied to the descriptor @TEMP.

6. Subscripted expressions when the the subscriptee and the indices satisfy particular conditions. The method of beating selection operators when the right operand is a subscripted expression is shown by the equivalence of expressions below. The expressions are listed in pairs; each pair is equivalent.

$Q{\downarrow}A[I1;I2;\ \ldots\ ;IN]$
$A[Q[1]{\downarrow}I1;Q[2]{\downarrow}I2;\ \ldots\ ;Q[N]{\downarrow}IN]$

$Q{\uparrow}A[I1;I2;\ \ldots\ ;IN]$
$A[Q[1]{\uparrow}I1;Q[2]{\uparrow}I2;\ \ldots\ ;Q[N]{\uparrow}IN]$

$\phi[Q]A[I1;I2;\ \ldots\ ;IN]$
$A[I1;I2;\ \ldots\ ;\phi IQ;\ \ldots\ ;IN]$

$Q\backslash A[I1;I2;\ \ldots\ ;IN]$
$(Q\backslash A)[J1;J2;\ \ldots\ ;JN]$

It is assumed here that the definition of Take and Drop is such that $v/(|Q)>\rho A$ is always zero. If the extended definition of Take and Drop is used then this condition would be tested for before the equivalent expression could be beaten. The indices for Take and Drop must be beatable expressions. The

Qth indicy must be beatable for Reverse. Transpose requires A to be beatable as well as Q being a permutation of (⍳⍴Q). The subscript J1 is identical to In where n is (Q⍳1). In general Jm is identical to In where n is (Q⍳m). For example, the following three expressions are equivalent:

```
(3 1 2)⌽A[I1;I2;I3]
((3 1 2)⌽A)[J1;J2;J3]
((3 1 2)⌽A)[I2;I3;I1]
```

4.4 The E Machine (EM)

The organization and operation of the EM more closely resembles conventional computers than any other component of the APLM. The EM moves scalar values between registers and memory, performs various primitive operations, such as add, on the values in the registers, and controls the mechanism for the indexing of arrays. A subset of EM instructions with a brief annotation to explain each follows:

EM Instruction Set

Load Scalar: S scalar

The scalar is an immediate operand of the instruction. The scalar operand is pushed to the VS with the tag ST denoting it is a scalar value.

Initialize Fetch Array: IFA descriptcr, MASK=00011

This instruction creates an FA instruction and replaces itself with the new instruction. FA instructions are not generated directly by the DM as IFA instructions are much more easily beaten. The DM instruction LDNF causes the creation of IFA instructions. The descriptor from the NT associated with the operand of the LDNF instruction is inserted into the IFA instruction, and the descriptors reference count incremented. Subsequent beating of the IFA instruction will cause a copy of the descriptor to be made and the operand of IFA changed to point to the new descriptor. To execute the IFA instruction the EM accesses the information in the descriptor as well as the IS to build an iteration control block, (ICB) at the top of the QS. The entries in the ICB are explained below. The EM pushes an entry to the ICB for every one value it finds in the boolean, vector MASK. MASK determines by bit position which entries of the IS will be used in building the ICB. The right-most value in MASK corresponds to the bottom entry in the stack.

Fetch Array: FA LINK=10,VBASE=700,SUM=164

This instruction is used to fetch elements from arrays and push them to the VS with the tag ST. LINK is the displacement from the FA instruction to the ICB. VBASE is

added to SUM to generate the memory address from which a value will be fetched.

Initialize Array Address: IA descriptor,MASK=00011

IA instructions are very similar to IFA instructions. They differ in that the DM instruction LDN generates them and that they in turn generate A instructions.

Fetch Array Address: A LINK=20,VBASE=950,SUM=164

This instruction generates a memory address and pushes it to the VS with the tag AT. Its operands are the same as the FA instruction.

Initialize J-Vector: IJ len,org,dir,MASK=00010

This instruction creates a J instruction and replaces itself with the new instruction. MASK is used to link the J instruction with an entry in the IS. The IS entry will have an ISMAX field equivalent to the operand len. The org and dir operands of IJ become the incr and dir operands of J. The curr operand of J is initialized to zero.

Fetch J-Vector: J curr,incr,dir,INX=2

The J instruction generates the scalar component of the J-vector it defines and pushes this value to the VS with tag ST. To generate the value the instruction tests the ISCH

field of the IS entry indexed by INX. If ISCH is one then ISCTR replaces curr and curr plus incr is pushed to the VS. If ISCH is zero then curr plus incr is pushed to the VS which is the same value as was previously pushed to the VS. The incrementing of J-vectors is linked to the VS to facilitate the generation of indices for subscripted arrays.

Operator: OP sao

The operand is one of the monadic or dyadic scalar arithmetic operators. The operation is applied to the top one or two values on the VS and the result left in place of the operand or operands.

Result Dimension: IRD descriptor,MASK=00011

This EM instruction is really a DM tag denoting that the dimension of a result can be found by inspecting the operands. Segment descriptors in the VS define for the DM deferred segments of code. The dimensions to be associated with these segments can be found by scanning backwards through them searching for IRD, IA or IFA instructions. IRD instructions are inserted when the IA or IFA instructions are not indicative of the resulting dimension. For instance, inner and outer products, reduction, compression and expansion are examples of operations in which the operands do not have the same dimension as the result. The

EM changes an IRD to a NIL instruction.

Load Segment Descriptor: SGV org,len,mode

The operands org, len, and mode describe an EM segment of code. Org is relative to the SGV instruction. It represents a displacement that must be subtracted from the current instruction address. Org is made into an absolute QS address and the segment descriptor pushed to the VS with the tag ST.

Jump: JMP link

The ISREL field is incremented by the amount link. Link is a signed integer.

Jump Zero: J0 link

The ISREL field is incremented by the amount link if the value on the top of the VS is zero. The VS is popped. JNC is a similar instruction but the VS is not popped.

Jump One: J1 link

Similar to J0 but the VS must be one. JN1 is similar to J1 but the VS is not popped.

Reduction: RED link

Push an entry to the VS with tag RT. The value field of the VS entry is irrelevant. When one of the operands of OP is applied to the VS and a VS entry has the tag RT the operation is not applied. The entry without the RT tag is used as a result. Effectively an RT entry is a place holder for dyadic operators. Once the reduction has started the RT entry will disappear. In the case where the expression, (+/⍳0), is encountered the DM takes the appropriate action as (⍳0) can not be handled by the EM. After pushing the entry to the VS a jump is made by incrementing LSREL by the value of link.

Mark and Iterate: MIT no operand

This instruction adds entries to the IS. Scalar values, marked by the tag ST, are popped from the VS. The absolute value minus one of each scalar is used to specify ISMAX in a new IS entry. ISCH is set to one and ISMRK to zero. If the VS value was positive then ISCTR and ISDIR are set to zero. If the value is negative then ISDIR is set to one and ISCTR to ISMAX. The value of ISMRK in the first entry created is changed to one. When an entry is popped from the VS and the tag is not ST then it must be SGT - a segment descriptor created by a SGV instruction. MIT uses the segment descriptor to push a new entry to the LS. The

ISIS is set to one so that the newly created IS will be iterated through and popped form the IS.

Pop: POP no operand

The top element is popped from the VS.

Initialize Segment Conditional:

ISC crg,len,mode,MASK=00010

ISC initializes to an SC instruction. The operands org, len, and mode describe an EM segment of code similar to the SGU instruction. The MASK will have only one non-zero bit. The position of this bit will determine the LINK value.

Segment Conditional: SC org,len,mode,LINK=3

The instruction uses the LINK value to test the relevant ISCH bit. If the bit is one then an entry is pushed to the LS envoking the segment of code specified by the other operands. The LINK operand is also a relative backward displacement to a group of XT entries in pushing the entry to the LS the LINK operand is made absolute and specifies the LSQP field. The LSQP field will be used by the Index Unit to locate the XT entries. If ISCH is zero then the following instruction is inspected. If it is one of XS or XC the LSREI is incremented so the instruction will

not be executed. Also the LINK field of the XS or XC instruction is followed to an XT entry. An XT entry is a pseudo-IS entry used for indexing subscripted arrays. The CH bit in the XT entry is set to zero indicating that the corresponding IS entry has ISCH set to zero.

Initialize Index Load: IXL MASK=00001

IXL initializes to an XL instruction. The MASK is applied to the IS and an INX value generated in the same way an INX value is generated for an ICB.

Index Load: XL INX=3

The INX value is used to access an IS entry. The ISCTR field is pushed to the VS with the tag ST.

Index Store: XS LINK=5

The LINK field is a relative backward displacement to an XT entry in the QS. The VS is popped and its value pushed to the index field of the XT entry. XT entries are explained under subscripted arrays.

Activate Segment: SG org,len,mode,LINK=3

The operands of SG are the same as those for SC. SG unconditionally pushes an entry to the LS similar to SC. The other functions done by SC are not done by SG.

The operation of the EM will be explained by examining the example below. The explanation will consist of two parts. In the first part, the simplest part, A and B will be one-element arrays. In the second part A and B will be multi-element arrays. Details relevant to indexing will be excluded from the first part and be the sole concern of the second.

$$Z\leftarrow((A*2)+(B*2))*.5$$

	IM Code			EM Code	
Address	Operation	Operand	QSADDR	Operation	Operands
300	IDS	.5	0	S	.5
301	IDS	2	1	S	2
302	LDNF	B	2	IFA	@B,MASK=00111
303	POWER		3	OP	PWR
304	IDS	2	4	S	2
305	LDNF	A	5	IFA	@A,MASK=00111
306	POWER		6	OP	PWR
307	ADD		7	OP	ADD
308	POWER		8	OP	PWR
309	LDN	Z	9	IA	@Z,MASK=00111
310	ASGN		10	OP	ASGN
			11	POP	

Assume A and B to be one-element, three-dimensional arrays. The DM envokes the EM by pushing the following entry to the top of the LS:

REL	ORG	LEN	CDE	IS	FN	NWT	QP
0	0	12	01	1	0	0	0

This entry initiates execution of the EM code in the example above. The first three instructions of EM code push .5, then 2, and finally B to the top of the VS. In the process of being executed the IFA instruction for B has been changed to an FA instruction. This transformation facilitates indexing. Initialization instructions, such as IFA, do not cause the LSREL field to be incremented. Consequently the FA instruction that replaces IFA is executed following the completion of the IFA instruction. It is the FA instruction which fetches the value of B. The following OP instruction pops the top two elements from the VS, performs the operation power, and pushes the result to the top of the stack. The next five instructions push values to the VS or perform the indicated arithmetic operation. The IA instruction is executed and changed to A. Again, A is executed immediately after IA. The A instruction pushes the memory address of Z to the VS. The following OP instruction pops the VS and uses the address to store the value now on the top of the VS. The stored value is left on top of the VS as the result of the operation ASGN. The following

instruction, POP, clears the value off of the VS. Following the execution of POP the LS overflows; that is, LSREL equals LSLEN and the top entry is popped from the LS. The APLM then returns control to the DM and continues execution.

For the second part assume A and B are three-dimensional arrays with dimensions 3, 5, 11. To calculate the result the EM code is embedded in a set of nested loops. Each loop is an entry in the IS. For the above example the IS would be initialized by the DM to:

	CTR	MAX	DIR	CH	MRK
	0	2	0	1	1
	0	4	0	1	0
TOP =>	0	10	0	1	0

As in the first part when the EM code shown above has been completely executed once, the LS overflows. At this point the IS field of the LS register is tested. If LSIS is zero then the LS is popped as the segment of EM code described by this LS entry is not embedded in an IS loop. If LSIS is one then LSREL is set to zero preparatory to reactivation of the segment of EM code. The IS is then stepped. To step the IS all the ISCH fields are zeroed. Then the CTR field of an IS entry is incremented by one and the CH field set to one. If the entry overflows - ISCTR equals ISMAX - then ISCTR is set to zero and the next lower entry in the stack stepped. The

stepping process starts with the entry at the top of the stack and continues down until an entry does not overflow or ISMRK is turned on in the entry. ISMRK marks the outermost loop. If this entry overflows then the IS overflows and the LS is popped. If, in stepping the IS, it does not overflow then the LS will cause the EM code to be re-executed. Notice that the ISCTR fields could be used as arguments to the polynomial access functions for the arrays A and B.

If we use the ISCTR fields to evaluate the polynomial access function in the above example we will find the difference between any two successive evaluations of the polynomial to be one. This suggests incrementation as a simpler and more efficient method of evaluating the polynomial. Incrementing by one is sufficient to index deferred expressions but not for beaten expressions. A more general method is evident in the execution of EM IFA instructions.

The instruction of QSADDR 2 in the example above is:

```
IFA     @B,MASK=00111
```

The @B operand is a pointer to the descriptor for B. MASK is a logical vector. Although only five of its values are shown there should be as many as the highest rank the implementation supports - generally 32. The one values select entries from the IS that are involved in indexing the

array B. In this example it selects the top three IS entries. MASK performs a useful function when the arrays are operands of the Outer Product operation. An IFA instruction is only executed once. In the process of executing, IFA the opcode and operands are changed. The instruction becomes:

```
FA      LINK=10,VBASE=700,SUM=164
```

The link field is a relative displacement to an iteration control block (ICB) pushed to the top of the QS. VBASE is the memory address of the values of B. SUM is the displacement from VBASE to the value indexed. All these fields are set up by the execution of the IFA instruction. The values for VBASE and the ICB were obtained from the MASK and the descriptor for B. The ICB contains values which are added to or subtracted from SUM. Using the ICB we can implement the more generalized incrementing function mentioned above. The ICB entries contain the following values:

QSADDR	Operation	Operands
12	NT	Q1=110,Q2=55,INX=0
13	NT	Q1= 44,Q2=11,INX=1
14	NLT	Q1= 10,Q2= 1,INX=2

NT and NLT are not opcodes but rather tags which identify the ICB entries as being associated with the IS. The alternative is QT and QLT tags that are used for subscripted

arrays which will be explained later. The INX operands index the Nth entry from the bottom of the IS. Thus INX=2 would index the entry marked TOP in the IS shown above and INX is relative to the bottom of the IS as the displacement from the bottom to any given entry remains constant. As entries may be pushed to the top of the IS after an INX value is set it must be relative to the bottom of the IS. INX=0 indexes the entry with ISMRK set to one. The Q2 operands are the coefficients of the polynomial access function. The Q1 operand is the product of Q2 and the ISMAX field selected by using the INX operand. Q1 and Q2 have more intuitively understandable interpretations. Q2 is the increment in the displacement needed to access a value whose indices differ from a previous value's by one in only one of the dimensions. For example the increment in displacement between B[0;0;0] and B[0;0;1] is 1, between B[0;0;0] and B[0;1;0] is 11, and between B[0;0;0] and B[1;0;0] is 55. Q2 is the decrement in displacement needed when the indices differ in only one dimension but take on their extreme values in that dimension. For example the decrement in displacement between B[0;0;10] and B[0;0;0] is 10, between B[0;4;0] and B[0;0;0] is 44, between B[2;0;0] and B[0;0;0] is 110, and between B[2;4;10] and B[0;0;0] is 110 + 44 + 10 = 164. Notice that the operand SUM in the FA instruction above has an initial value of 164. The purpose of this

initialization will become apparent in the explanation of the FA instruction below.

All EM instructions with mnemonics that start with an "I" such as IFA, are initialization instructions. When executed they initialize an associated instruction and suppress the incrementation of the LS. Thus the next instruction executed is the newly initialized instruction.

When the FA instruction is executed it invokes the Index Unit. The Index Unit alters the value of SUM and returns. The FA instruction then adds VBASE and SUM to generate a memory address from which it fetches a value. This value is then pushed to the top of the VS.

The Index Unit used the LINK operand to access the ICB. It uses the INX operand of the first entry - QSADDR 12 - to index an entry of the IS. As ISCH is one in this entry, the Index Unit will use the ICB entry to increment or decrement SUM. As the IS entry appears to have overflowed, ISCTR is zero, Q1 is subtracted from SUM. The Index Unit continues with the other ICB entries in a similar manner. On termination it has decremented SUM to zero. Before this FA instruction is executed again the LS will have overflowed and the IS stepped. In stepping the IS all the ISCH entries will have been zeroed. The IS entry at the top of the stack

will then be stepped and only this ISCH field reset to one. Thus the Index Unit will do nothing with first two ICB entries as ISCH is zero. The third entry, for which ISCH is one and ISCTR is not zero, will cause the incrementation of SUM by Q2.

A question remains as to why SUM was initialized to 164 rather than zero. By initializing in this fashion the logic to handle subscripted arrays is simpler. For the array A[I1;I2] the values of I1 or I2 are fetched if the associated ISCH is one. By initializing all the ISCH fields to one the indices are fetched on the first pass through the EM code by the Index Unit. Thus the logic to fetch indices in the Index Unit need not be duplicated in the IFA instruction.

4.4.1 Subscripted Arrays

The handling of subscripted arrays by the EM is intended to parallel as closely as possible that of non-subscripted arrays. The array accessing mechanism will have to be augmented so that the indices are used as arguments to the polynomial access function rather than the values of ISCTR. However the IS still remains as an integral part of the accessing mechanism. Its function is illustrated with the aid of the following expression:

T + A[B[C[I;];];]

Assume that T is not a scalar or one-element array and that the dimension of I is 1, of C is 2 3, of B is 4 5, and of A is 6 7. If the add operation has conformable operands then T must have dimension 1 3 5 7. The values of I would be fetched using the IS which would have the following entries:

IS: CTR	MAX	DIR	CH	MRK	ENTRY NO.
0	0	0	0	1	1
0	2	0	0	0	2
0	4	0	0	0	3
0	6	0	0	0	4

Note: The Entry Number shown is for ease of reference in the text below and is not part of the IS.

Notice that the values of I can be accessed using the first IS entry and that when this entry overflows we will have fetched all the values of I necessary. The IS can not be used to access C[I;] directly but the dimension of C[I;] is 1 3 which is the same as the first two entries of the IS. Thus when the first two entries of the IS overflow all the values of C[I;] will have been fetched. When the first three entries overflow B[C[I;];] will have been fetched and when all four entries overflow A[B[C[I;];];] will have been fetched. The IS will perform two functions in the accessing of arrays:

1. When the relevant ISCH field is one a new value will be fetched from a subscript.
2. When all the values of a subscript have been fetched the relevant IS entry will overflow.

The function of the ISCTR and ISMAX fields of the IS are replaced by XT or pseudo-IS entries in the QS for subscripted arrays. XT entries have the following fields:

```
XT    index,limit,change,GN=3
```

The first three operands parallel the ISCTR, ISMAX and ISCH fields of the IS. Group Number (GN) appears only in the first entry in a group of entries and determines the number of entries in the group. XT is used in conjunction with the ICB to access array elements. When the Index Unit (IU) is invoked it finds that the LSQP field is non-zero. The IU uses LSQP as a pointer to the XT entries. The IU accesses the first ICB entry and uses the INX value to select the relevant XT entry. If the change field of the XT entry is zero the IU goes onto the next ICB entry. Otherwise it subtracts from the operand SUM in the FA instruction which invoked the IU the Q1 value. It then adds to SUM the product of Q2 and the index field of the XT entry. This product is then used to respecify Q1. Consider what occurs when the IU is first invoked for an FA instruction. SUM has been initialized to the displacement of the last element of

the array. Subtracting the Q1 values will decrement SUM to zero. Adding the product increments SUM to element specified by the indicy. Replacing Q1 by the product is the simplest way of having SUM reset to zero next time the IU is invoked. The IU processes the remaining entries in the ICB in a similar fashion.

EM instructions are used to alter the values of index and change in XT entries. The two pairs of instructions used are:

```
1.  ISC    org,len,mode,MASK=00001
    XS     INX=4
2.  IXI    MASK=00010
    XS     INX=4
```

The first pair is used for fetching and storing subscripts in XT entires. The second pair is used to retrieving ISCTR and storing it in XT. This will occur when there is an empty subscript such as the first dimension in the expression A[I;]. An ISC instruction is used rather than IFA as the subscripts may be expressions rather than simple operands. In addition ISC provides for the specifying of the LSQP so that the IU will function correctly.

Referring to the expression above, A[B[C[I;];];], the EM code to fetch the values of I is:

```
QSADDR    Operation    Operands
  1          IFA       @I,MASK=00001
```

The EM code to fetch the values of C[I;] is:

```
QSADDR    Operation    Operands
  *  Code to fetch the value of I
  2          IFA       @C,MASK=00011
  3          XT        0,1,1,GN=2
  4          XT        0,2,1
  5          ISC       4,1,1,MASK=00001
  6          XS        LINK=3
  7          IXL       MASK=00010
  8          XS        LINK=4
  9          SG        7,1,1,LINK=6
```

To invoke this segment of code an entry is pushed to the LS which defines QS addresses 5 through 9. The ISC instruction at QSADDR 5 causes the IFA instruction at QSADDR 1 to be executed. The following XS instruction stores the value of I in the XT entry at QSADDR 3. Notice that the index fields of the XT entries are zero prior to XS instructions being executed. The following IXL, XS pair push the value of ISCTR to the XT entry at QSADDR 4. The SG instruction at QSADDR 9 executes the IFA instruction at QSADDR 2 and a value of C is pushed to the VS. The EM code to fetch the values of B[C[I;];] is:

```
QSADDR    Operation    Operands
  *  Code to fetch C[I;]
  10          IFA        @B,MASK=00111
  11          XT         0,3,1,GN=2
  12          XT         0,4,1
  13          ISC        8,5,1,MASK=00001
  14          XS         LINK=3
  15          IXL        MASK=00100
  16          XS         LINK=4
  17          SG         7,1,1,LINK=6
```

This segment of code executes similar to the above segment. However, the ISC instruction at QSADDR 13 invokes the segment of code at QS addresses 5 through 9. The EM code to fetch the values of A[B[C[I;];];] is:

```
QSADDR    Operation    Operand
  C           JMP        21
  *  Code to fetch B[C[I;];]
  18          IFA        @A,MASK=01111
  19          XT         0,5,1,GN=2
  20          XT         0,6,1
  21          ISC        8,5,1,MASK=00001
  22          XS         LINK=3
  23          IXL        MASK=01000
  24          XS         LINK=4
  25          SG         7,1,1,LINK=6
```

The LS entry which invoked this segment has LSORG set to zero and LSLEN set to 25. The LSIS bit is one, thus this segment will be repeated until the IS overflows. The first dimension of A[B[C[I;];];] is 7. Thus the instructions at QS addresses 0, 21, 22, 23, 24, 25 are executed 7 times before the IS overflows.

4.4.2 Reduction

An expression containing a Reduction operator produces a result of rank one less than the operand of the Reduction. The DM initializes the IS to represent the rank and dimensions of the result. Thus, in order to use the IS to access the operand of Reduction, instructions are inserted in the EM program segment to push an entry to the IS. This is illustrated by the following example:

APL Expression: A←+/B

QSADDR	Operation	Operands
0	RED	LINK=3
1	IFA	@B,MASK=00111
2	CP	ADD
3	SGV	2,2,1
4	S	-5
5	MIT	
6	IRD	@TEMP,MASK=00011

```
7          IFA        @A,MASK=00011
8          OP         SUB
```

Assume A has dimensions 2 3 and B 2 3 5. The IS entry for this segment invokes the instructions at QS addresses 0 through 8. The first instruction pushes an entry to the VS with tag FT and branches to QSADDR 3. The SGV instruction pushes a segment descriptor for QS addresses 1 and 2 to the VS with tag SGT. The following instructions pushes value minus five to the VS with tag ST. Five is the value of the third co-ordinate in the dimension of B. The minus sign will indicate to the following MIT instruction that the ISDIR field is to be one and consequently the reduction process will decrement along the third co-ordinate of B. The MIT instruction pops all entries with tag ST from the VS. As it pops these values it uses them to specify ISMAX in IS entries it is constructing. MIT then pops the SGT entry from the VS and uses it to push an entry to the IS. The new LS entry under IS control iterates through the instructions at QS addresses 1 and 2 five times and is then popped as the IS overflows. This calculates one scalar component of +/B. This scalar component is then subtracted from a scalar component of A. The LS iterates through the QS addresses 0 through 8 six times before the IS overflows and A-+/B is calculated.

In the more general case where an index is included in the Reduction expression such as +/[K]B the DM constructs EM code to calculate a different but equivalent expression. If B is a rank 4 array then the expressions below are equivalent:

+/[1]*B*	is equivalent to	+/(4 1 2 3)⍉*B*
+/[2]*B*	is equivalent to	+/(1 4 2 3)⍉*B*
+/[3]*B*	is equivalent to	+/(1 2 4 3)⍉*B*
+/[4]*B*	is equivalent to	+/(1 2 3 4)⍉*B*
+/[4]*B*	is equivalent to	+/*B*

In general +/[*K*]*B* is equivalent to +/*A*⍉*B* where A is:

(⍳*K*-1),(⌈/⍴⍴*B*),((*K*-1)+⍳(⍴⍴*B*)-*K*)

As the Transpose operator is beaten by the DM the general case is equivalent to the particular case illustrated above.

4.4.3 Inner and Outer Products

An expression containing an Outer Product produces a result of greater rank than either of the operands. Like Reduction the IS is initialized to represent the rank and dimension of the result. As the dimension of the result is the catenation of the dimensions of the operands the entries in the IS can be used directly to fetch the operands. All that is needed is to alter the masks of the IFA instructions that fetch the operands. Consider the following example:

API Expression: $A \leftarrow B \circ .\times C$

QSADDR	Operation	Operands
0	IFA	@C,MASK=00011
1	IFA	@B,MASK=01100
2	OP	MULT
3	IA	@A,MASK=01111
4	ASGN	
5	PCP	

Assume the dimension of B is 3 5 and C is 7 9. The dimension of A is then 3 5 7 9. The IS will contain:

IS: CTR	MAX	DIR	CH	MRK	ENTRY NO
0	2	C	0	1	1
0	4	0	0	0	2
0	6	0	0	0	3
0	8	C	0	0	4

The MASK for array C selects IS entries 3 and 4 to fetch the values of C. The MASK for B is shifted left by the rank of C thus selecting entries 1 and 2. The MASK for A is the logical or of the masks of the operands and thus selects all the entries of the IS. As each value of C is fetched $\times/\rho B$ times and each value of B is fetched $\times/\rho C$ times, if the operands are expressions they are calculated to temporary space before the Outer Product is deferred.

To calculate Inner Products the DM constructs EM code

to calculate a different but equivalent expression. If the original expression is $B+.\times C$ then the equivalent expresion is $+/A\circ.\times C$ where A is the following:

$$(\iota^{-}1+\rho\rho B),(2\rho\lceil/\iota(\rho\rho B)+(\rho\rho C)-1),(^{-}1+\rho\rho B)+\iota^{-}1+\rho\rho C$$

For example if the dimension of B is 2 3 4 and C is 4 5 6 then the DM generates the expression:

$$+/((1\ 2),(5\ 5),(3\ 4))\circ.\times C$$

This expression is beaten then handled in the same manner as the Reduction expressions discussed above.

The purpose of the APLM is to achieve efficiency in the execution of APL expressions by beating and defferal. Beating reduces the number of operations required to evaluate an expression. Deferral makes more efficient use of low speed core memory. To achieve these ends the hardware design is made to closely resemble the design of the language.

CHAPTER V

Conclusion

This Chapter reviews the main points of the previous Chapters and analyzes the result.

Iverson[14] originally designed APL to be a language for the description and communication of algorithms. In implementing APL the I/O facility, tree structures, and many of the operators were omitted. In addition much of the notation and character set were altered. Despite these omissions the resulting language is a highly consistent, mathematically oriented language. However there are faults. Subscripting was not fully generalized to arrays and the syntax is context dependent. Subscripting can be generalized to arrays by the addition of a mixed-data type. The mixed-data type, being similar to parameter lists in other languages, will also allow more than two operands to be explicitly passed to a function. Operands were not passed to functions by the same mechanism as operands are passed to primitive operators. By passing operands to functions by name we can make consistent the facility of functions augmenting the primitive function set. The Branch operator was deleted because its function could better served by the control structures IF, THEN, ELSE and REPEAT,

END. The new syntax adheres to Iverson's criteria of clearly exhibiting the constraints on the sequence in which operations are performed. In addition it implements the method of leading decisions and allows no alternatives. The operators Encode, Decode, Index Generator and Index Of are extended to arrays from their previously restricted definitions.

A new operator, Rename, and the addition of an I/O facility were discussed in Chapter III. Rename was added to allow variables, especially "parameter lists" or variables of type mixed-data, to be renamed and manipulated without the necessity of duplicating the arrays involved. The facility provided by Rename is consistent with the facility provided by function invocation - that of values being known by another name. An I/O facility for sequential files was added to increase the usefulness of the language. It satisfies Iverson's criteria that it be concise, consistent, mnemonic and economical of symbols to a greater degree than the ad hoc implementing of I/O functions and operators. The I/O facility was designed to be reasonably simple and straight-forward to implement. It does not provide the full complement of facilities such as indexed or shared files that would be useful.

Improvements to the implementation consist of

modifications to Abrams proposal for an APL machine. Abrams' APLM is a stack oriented machine divided into three parts. The CM converts APL text to a more readily interpreted DM code. DM code is in an opcode-operand form that resembles the machine code of many computers. The DM interprets the DM code, analyzing it into deferrable and beatable expressions. The results of the analyses is executable EM code. The EM code has well defined operations and operands and is executed directly.

A subset of the EM was implemented in IBM/360 Assembler Langauge. The subset worked and demonstrated that the fundamental design of Abrams' machine is valid.

The advantages of the APLM are the following:

1. DM code can be interpreted efficiently.
2. Deferred expressions reduce the number of references to intermediate speed storage.
3. Beating reduces the number of computations necessary to evaluate an expression.
4. The stack discipline of the APLM organizes frequently used data into the high speed storage or stacks.

The fourth point needs some qualification. If the high speed storage is an order of magnitude faster than intermediate speed storage then the stack discipline is

clearly advantageous. However the advent of scratch-pad memories and other technological advances reduces the advantage of the APLM. In the simulation of the APLM on a machine with no stack hardware deferred expressions are a disadvantage, unless beating occurs to reduce the number of calculations so as to gain an overall advantage. Some of the disadvantages of the APLM are:

1. The generation of a precise error indication for errors such as division by zero in a deferred expression is an involved procedure.
2. The number and size of the various registers makes task switching in a multi-programming environment unattractive.
3. In order to gain the advantages mentioned above the IS, LS, QS, and VS should be implemented in high speed register storage which may be costly. In addition the logic of the DM and EM would be microprogrammed using a considerable amount of storage.

The overall advantage of the APLM is dependent on changing technology. The introduction of scratch-pad memories to conventional implementations may reduce the advantage of the APLM to the point where its extra cost is unwarranted.

Appendix A

Summary of APL Operators

Monadic form fB		f	Dyadic form AfB	
Definition or example	Name		Name	Definition or example
$+B \leftrightarrow 0+B$	Plus	+	Plus	2+3.2 ↔ 5.2
$-B \leftrightarrow 0-B$	Negative	-	Minus	2-3.2 ↔ ¯1.2
$\times B \leftrightarrow (B>0)-(B<0)$	Signum	×	Times	2×3.2 ↔ 6.4
$\div B \leftrightarrow 1\div B$	Reciprocal	÷	Divide	2÷3.2 ↔ 0.625
B | $\lceil B$ | $\lfloor B$ ¯3.14 | 4 | 3 ¯3.14 | ¯3 | ¯4	Ceiling	⌈	Maximum	3⌈7 ↔ 7
	Floor	⌊	Minimum	3⌊7 ↔ 3
$*B \leftrightarrow (2.71828..)*B$	Exponential	*	Power	2*3 ↔ 8
$\bullet *N \leftrightarrow N \leftrightarrow *\bullet N$	Natural logarithm	⍟	Logarithm	$A\bullet B \leftrightarrow$ Log B base A $A\bullet B \leftrightarrow (\bullet B)\div\bullet A$
|¯3.14 ↔ 3.14	Magnitude	|	Residue	Case | $A\|B$ $A\neq 0$ | $B-(\|A)\times\lfloor B\div\|A$ $A=0, B\geq 0$ | B $A=0, B<0$ | Domain error
!0 ↔ 1 $!B \leftrightarrow B\times !B-1$ or $!B \leftrightarrow$ Gamma$(B+1)$	Factorial	!	Binomial coefficient	$A!B \leftrightarrow (!B)\div(!A)\times !B-A$ 2!5 ↔ 10 3!5 ↔ 10
$?B \leftrightarrow$ Random choice from ιB	Roll	?	Deal	A Mixed Function (See Table 3.8)
$\circ B \leftrightarrow B\times 3.14159...$	Pi times	○	Circular	See Table at left
~1 ↔ 0 ~0 ↔1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	

A	B	$A\wedge B$	$A\vee B$	$A⍲B$	$A⍱B$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

<	Less	Relations
≤	Not greater	Result is 1 if the
=	Equal	relation holds, 0
≥	Not less	if it does not:
>	Greater	3≤7 ↔ 1
≠	Not Equal	7≤3 ↔ 0

$(-A)\circ B$	A	$A\circ B$
$(1-B*2)*.5$	0	$(1-B*2)*.5$
Arcsin B	1	Sine B
Arccos B	2	Cosine B
Arctan B	3	Tangent B
$(¯1+B*2)*.5$	4	$(1+B*2)*.5$
Arcsinh B	5	Sinh B
Arccosh B	6	Cosh B
Arctanh B	7	Tanh B

Table of Dyadic ○ Functions

APL Scalar Functions

ρA	ρB	ρAf.gB	Conformability requirements	Definition Z←Af.gB
				Z←f/AqB
	V			Z←f/AgB
U				Z←f/AgB
U	V		U=V	Z←f/AqB
	V W	W		Z[I]←f/AgB[;I]
T U		T		Z[I]←f/A[I;]gB
U	V W	W	U=V	Z[I]←f/AgB[;I]
T U	V	T	U=V	Z[I]←f/A[I;]gB
T U	V W	T W	U=V	Z[I;J]←f/A[I;]gB[;J]

APL Inner Products

ρA	ρB	ρA∘.gB	Definition Z←A∘.gB
			Z←AgB
	V	V	Z[I]←AgB[I]
U		U	Z[I]←A[I]gB
U	V	U V	Z[I;J]←A[I]gB[J]
	V W	V W	Z[I;J]←AgB[I;J]
T U		T U	Z[I;J]←A[I;J]gB
U	V W	U V W	Z[I;J;K]←A[I]gB[J;K]
T U	V	T U V	Z[I;J;K]←A[I;J]gB[K]
T U	V W	T U V W	Z[I;J;K;L]←A[I;J]gB[K;L]

APL Outer Products

Name	Sign[1]	Definition or example[2]
Size	⍴A	⍴P ↔ 4 ⍴E ↔ 3 4 ⍴5 ↔ ⍳0
Reshape	V⍴A	Reshape A to dimension V 3 4⍴⍳12 ↔ E 12⍴E ↔ ⍳12 0⍴E ↔ ⍳0
Ravel	,A	,A ↔ (×/⍴A)⍴A ,E ↔⍳12 ⍴,5 ↔ 1
Catenate	V,V	P,⍳2 ↔ 2 3 5 7 1 2 'T','HIS' ↔ 'THIS'
Index[3,4]	V[A] M[A;A] A[A;...;A]	P[2] ↔3 P[4 3 2 1] ↔7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ ABCD EFGH IJKL
Index generator[3]	⍳S	First S integers ⍳4 ↔ 1 2 3 4 ⍳0 ↔ an empty vector
Index of[3]	V⍳A	Least index of A in V, or 1+⍴V P⍳3 ↔2 5 1 2 5 P⍳E ↔ 3 5 4 5 4 4⍳4 ↔ 1 5 5 5 5
Take	V↑A	Take or drop ⌈V[I] first (V[I]≥0) or last (V[I]<0) elements of coordinate I 2 3↑X ↔ ABC EFG ¯2↑P ↔ 5 7
Drop	V↓A	
Grade up[3,5]	⍋A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[3,5]	⍒A	⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 ↔ 1 0 1⌿E 9 10 11 12
Expand[5]	V\A	1 0 1\⍳2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse[5]	⌽A	⌽X ↔ DCBA HGFE LKJI ⌽[1]X ↔ ⊖X ↔ IJKL EFGH ABCD ⌽P ↔ 7 5 3 2
Rotate[5]	A⌽A	3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ BCDA EFGH LIJK
Transpose	V⍉A	Coordinate I of A becomes coordinate V[I] of result 2 1⍉X ↔ AEI BFJ CGK DHL 1 1⍉E ↔ 1 6 11
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∈A	⍴W∈Y ↔ ⍴W 0 1 1 0 E∈P ↔ 1 0 1 0 P∈⍳4 ↔ 1 1 0 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	?S	W?Y ↔ Random deal of W elements from ⍳Y

1. Restrictions on argument ranks are indicated by: *S* for scalar, *V* for vector, *M* for matrix, *A* for Any. Except as the first argument of *S⍺A* or *S[A]*, a scalar may be used instead of a vector. A one-element array may replace any scalar.

2. Arrays used in examples: *P* ↔ 2 3 5 7 *E* ↔ 1 2 3 4 / 5 6 7 8 / 9 10 11 12 *X* ↔ *ABCD* / *EFGH* / *IJKL*

3. Function depends on index origin.

4. Elision of any index selects all along that coordinate.

5. The function is applied along the last coordinate; the symbols ⌿, ⍀, and ⊖ are equivalent to /, \, and ⌽, respectively, except that the function is applied along the first coordinate. If [*S*] appears after any of the symbols, the relevant coordinate is determined by the scalar *S*.

Notes for Mixed Functions

References

1. Abrams, P. S. An APL Machine. Report No. SLAC-114, Stanford Linear Accelerator Centre, Standford California, 1970.
2. Anger, Arthur I. Miscellanea. APL Quote-Quad, Vol. 2, No. 3, (1970), pp. 21.
3. Baumann, R., Feliciano, M., Bauer, F. L., Samelson, K. "Introduction to ALGOL." Prentice-Hall, Inc., Englewocd Cliffs, N. J., 1964.
4. Berry, P. C., Bartoli, G., Dell'Aquila, C., Spadavecchia, V. N. APL and insight. A strategy for teaching. In "Colloque API." IRIA, Rocquecourt, France, 1971, pp. 251-272.
5. Berry, p. APL/1130 Primer. Form No. C20-1697-0, IBM, White Plains, New York.
6. Bingham, Harvey W. Use of APL in microprogrammable machine modeling. Burroughs Corporation, Paoli, Pennslyvania.
7. Blankinship, W.A. An index catenator for APL. Unpublished notes of W. S. Adams, University of Alberta, Edmonton, Alberta.

8. Breed, L. M., Lathwell, R. H. The Implementation of APL/360. ACM Symposium on Experimental Systems for Interactive Applied Mathematics, IBM Watson Research Centre, Yorkton Heights, New York, 1967.

9. Breed, Larry. The APL PLUS file subsystem. APL Quote-Quad, Vol. 2, No. 3, (1970), pp. 4-6

10. Breed, Larry. Generalizing APL scalar extension. APL Quote-Quad, Vol. 2, No. 6, (1971), pp. 5-7.

11. Charmonman, S. A generalization of APL array-oriented concept. APL Quote-Quad, Vol. 2, No. 3, (1970), pp. 13-17.

12. Falkoff, A. D., Iverson, K. E. APL/360: User's Manual. IBM, T. J. Watson Research Centre, 1968.

13. Hassitt, A., Lageshulte, J. W., Lyon, L. E. A microprogrammed APL machine. In "Colloque APL." IRIA, Rocquencourt, France, 1971, pp. 375-382.

14. Iverson, Kenneth E. "A Programming Language." Wiley, New York, 1962.

15. McFarland, Clay. A language-oriented computer design. AFIPS conf. proc. FJCC, (1970), Vol. 37, pp. 629-640.

16. Mills, Harlen. Top down programming in large systems. In Rustin, Randall. (Ed.), "Debugging Techniques in Large Systems." Prentice-Hall, Inc., Englewood Cliffs, N. J., 1971, pp. 41-55.

17. Ryan, Jim. Generalized lists and other extensions. APL Quote-Quad, Vol. 3, No. 1, (1971), pp. 8-10.

18. Wulf, W. A., Russell, D. B., Habermann, A. N. BLISS: A language for systems programming. CACM, Vol. 14, No. 12, (1972), pp. 780-790.

19. Zaks, Rodney. Microprogrammed APL implementation. SIG MICRO Newsletter, Vol. 2, No. 3, (1971), pp. 5-6.

END OF REEL

ARC-30