SLAC-114

An APL Machine

Philip Abrams

SLAC Report No. 114

February 1970

AEC Contract AT(04-3)-515

STANFORD LINEAR ACCELERATOR CENTER
Stanford University • Stanford, California

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

SLAC-114
UC-32
(MISC)

AN APL MACHINE

PHILIP S. ABRAMS

STANFORD LINEAR ACCELERATOR CENTER

STANFORD UNIVERSITY

Stanford, California

PREPARED FOR THE U.S. ATOMIC ENERGY COMMISSION UNDER CONTRACT NO. AT(04-3)-515

February 1970

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Virginia 22151.
Price: Full size copy $3.00; microfiche copy $ .65.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

ABSTRACT

This dissertation proposes a design for a machine structure which is appropriate for APL and which evaluates programs in this language efficiently.

The approach taken is to study the semantics of APL operators and data structures rigorously and analytically. We exhibit a compactly representable standard form for select expressions, which are composed of operators which alter the size and ordering of array structures. In addition, we present a set of transformations sufficient to derive the equivalent standard form for any select expression. The standard form and transformations are then extended to include expressions containing other APL operators.

By applying the standard form transformations to storage access functions for arrays, select expressions in the machine can be evaluated without having to manipulate array values; this process is called beating. Drag-along is a second fundamental process which defers operations on array expressions, making possible simplifications of entire expressions through beating and also leading to more efficient evaluations of array expressions containing several operations.

The APL machine consists of two separate sub-machines sharing the same memory and registers. The D-machine applies beating and drag-along to defer simplified programs which the E-machine then evaluates. The major machine registers are stacks, and programs are organized into logical segments.

The performance of the entire APL machine is evaluated analytically by comparing it to a hypothetical naive machine based upon presently-available implementations for the language. For a variety of problems examined, the APL machine is the more efficient of the two in that it uses fewer memory accesses, arithmetic operations, and temporary stores; for some examples, the factor of improvement is proportional to the size of array operands.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks and deep gratitude to my dissertation advisor, Harold Stone, whose unselfish good counsel and understanding have been essential to the successful completion of this work. The other members of my reading committee, Bill Miller, Bill McKeeman, and Ed Davidson, have each contributed much appreciated time and energy towards improving the final form of this thesis. Larry Breed provided valuable help with his detailed readings and criticisms of the material in Chapter II. My friend and fellow student Sheldon Becker has been a kindred spirit during the vicissitudes of graduate student life. Thanks to Ken Iverson and Adin Falkoff who over the years have helped to imbue me with the "spirit of APL," and to Ed McCluskey, whose financial support enabled me to finish this work.

I dedicate this thesis to Life and Living, which begin anew each day.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Page

Chapter IV

Chapter V

LIST OF FIGURES

Page

Chapter IV

Chapter V

CHAPTER I

INTRODUCTION

an optimist is a guy that has never
had much experience

Don Marquis, archy and mehitabel

The electronic digital computer has progressed from being a dream, to an esoteric curiosity, to its present pervasive and indispensable role in modern society. Over the years, man's uses of computers have become increasingly sophisticated. Of particular importance is the use of high-level programming languages which have made machines more accessible to problem-solvers.

In general, the use of problem-oriented programming languages requires a relatively complex translation process in order to present them to machines. Although this can be done automatically by compilers, there is a wide gap to bridge between the highly-structured concepts in a programming language such as ALGOL, PL/I, or APL and the relatively atomic regime of today's computers. In effect, there exists a mismatch between the kinds of tasks we want to present to machines and the machines themselves. One possible way to eliminate this difference is to investigate ways of structuring machines to bring them closer to the kinds of problems people wish to solve with them.

A. A Programming Language

A particular programming language in which this mismatch with contemporary machines is especially obvious is APL, based on the work of K. E. Iverson (Iverson [1962]). APL is a concise, highly mathematical programming language designed to deal with array-structured data. APL programs generally contain expressions with arrays as operands and which evaluate to arrays, while most other languages require that array manipulations be expressed element-by-element. To complement its use of arrays as operands, APL is rich in operators which facilitate array calculations. Also, it is highly consistent internally both syntactically and semantically, and hence could be called "mathematical". Because of its use of structured data and its set of primitives which are quite different from those of a classical digital computer, APL does not fit well onto ordinary machines. It is possible to do so, and interpreters have been written for at least three different machines (Abrams [1966]; Berry [1968]; Pakin [1968]). Finally, because of its mathematical properties, it is possible to discuss the semantics of the language rigorously and to derive significant formal results about expressions in the language.

B. The Problem

The problem considered in this dissertation is to design a machine structure which is appropriate to APL. "Machine structure" here means a general functional scheme and not a detailed logical design. The expected result is not a set of specifications from which a circuit designer could produce a working device, but rather a superstructure into which the features of the language fit cleanly. Thus, this design must in some sense be natural for the language. For example, the primitive operations and data structures should include those of APL. In addition, the machine should take advantage of all available information in order to execute programs as efficiently as possible. We use the word "machine" in a very broad sense: what it really means here is "algorithm" and not necessarily any particular physical device. Such a machine could be implemented as a conventional program or as a hardwired device or as a microprogram in an appropriate system. For the purposes of this work, it doesn't really matter.

"APL" means any programming language which includes the semantics of APL\360 (Pakin [1968]). We shall not be concerned with the particular syntax of APL, although this currently appears to be the best way to represent the semantic ideas of the language. In short, the machine should be able to handle array-structured data with ease and should be able to evaluate functions on such data using the operators of APL as basic primitives.

The approach taken is to invest a considerable amount of effort in the analysis of the mathematical properties of the operators and data structures of APL and to exploit these results in the design of the machine. Thus, a major part of this work will be dedicated to a rigorous, mathematical investigation of APL expressions. This study is contained in Chapter II. In Chapter III, the work of Chapter II is related to the design of a machine, and the design goals are set forth in detail. Chapter IV discusses the proposed machine design, and Chapter V is an evaluation of the machine with respect to the goals of Chapter III.

It should be emphasized that the goal of designing an APL machine is a rather broad one. Although there are clearly practical applications of such a design, that is not the major focus of this work. Rather, we hope that by investigating this language and machine in detail, it will be possible to learn something about the basic processes in computing and find ways of reflecting these processes in a machine structure. The results summarized in Chapter VI and the new research problems suggested by this work indicate that this goal has been fulfilled.

C. Historical Perspective

For the purposes of this dissertation, we are primarily interested in previous work in the area of language-directed machine design (McKeeman [1967]; Barton [1965]). To some extent, all machine design can be considered to be language-directed, in that one wishes to implement some particular (machine) language in a piece of hardware. However, let us consider only the class of machines which might better be called "higher language inspired"; that is, machines which are based in some way on languages capable of expressing concepts at a higher level than are normally associated with assembly code.

The first such machine was reported in 1954, and was a relay device capable of directly evaluating logical expressions (Burks, Warren, and Wright [1954]). In addition, this machine used input in parenthesis-free (Polish) notation, thus doubling its historical interest. The logic machine typifies one major class of language-inspired machine designs in that its machine language is identical to the high-level source language. The other major class of language-inspired designs is more concerned with the processing of the semantics of the source language, rather than direct acceptance of the exact language by the machine. In fact, most designs fall between the two extremes, as even those which accept the source language directly do some preliminary transformations on it to produce a simpler intermediate language.

Other language-inspired machines accepting source language directly include an ALGOL 60 machine (Anderson [1961]), two FORTRAN machines (Bashkow, Sasson and Kronfeld [1967]; Melbourne and Pugmire [1965]), the ADAM machine, based on a special symbol-oriented language (Mullery, Schauer and Rice [1963]; Meggitt [1964]), and a machine for EULER, a generalization of ALGOL (Weber [1967]). Of these devices, some were to be implemented in hardware (e.g., Bashkow _et al._; Mullery _et al._) while others were implemented in microprogram (Meggitt; Weber).

Machines which are more concerned with semantic processing to the extent that their machine languages are significantly different from a higher-level language include the Burroughs B5000 (Barton [1961]; Burroughs [1963]) which is essentially an ALGOL machine, a PL/I machine (Sugimoto [1969]) and the Rice University computer (Iliffe and Jadeit [1962]). Current work in this area includes a PL/I machine (Wortman [1970]) and a micro-computer capable of emulating high-level processes easily (Lesser [1969]).

Most of these efforts are not directly relevant to the work in this dissertation and are thus reported here only for completeness. The common aspect of all these designs is that they are concerned with the processing of more highly organized information and programs than are found in the conventional von Neumann type architectures. Most of them include generalized addressing schemes using some modification of descriptors, as well as at least one stack.

Although the Burks, Warren, and Wright machine was the first to use Polish notation as a machine language, the first commercially produced devices to do so apparently were the English Electric KDF9 (Davis [1960]) and the Burroughs B5000. Both of these machines included stacks. Other related efforts not yet mentioned are two machines based on lower-level machine languages, but intended to deal with high-level primitives. One of these (Iliffe [1968]) is based on extensive use of descriptor logic for both programs and data, while the other (Myamlin and Smirnov [1968]) is somewhat more closely oriented toward higher-level languages. The latter, in particular, does run-time evaluation of infix arithmetic expressions.

Aside from the work of Burks *et al.*, none of the designs in the literature seem to be derived from explicit mathematical analysis of their input languages. Further, except for simulations or actual performance, none of the papers in the literature present satisfactory evaluations of their designs. This is not to say that the designs are not satisfactory: to the contrary, the success of the Burroughs family of computers and the KDF9 show that language-inspired designs are a viable approach to the development of new machines. On the other hand, nobody seems to have established exactly how viable such designs really are.

D. Conclusion

Having briefly reviewed the developments of language-inspired machine design to date, they can now be left in the background. The present approach is different from those in the past in that it is based on a mathematical analysis of the semantics of the source language. Also, the evaluation of the resulting design is analytic, and gives a clear comparison of this APL machine to other similar devices. There are, of course, similarities to the designs of the past. In particular, the use of program segments, data descriptors, and stacks is not novel in itself, although the machine developed here is substantially different from those mentioned in the last section.

"The thing can be done," said the Butcher, "I think.
    The thing must be done, I am sure.
The thing shall be done! Bring me paper and ink,
    The best there is time to procure."

L. Carroll, The Hunting of the Snark

CHAPTER II

MATHEMATICAL ANALYSIS OF APL OPERATORS

This chapter examines the mathematical properties of some of the APL operators. Mathematical definitions of the operators are given from which it is possible to deduce their properties. We show that there is a standard form for expressions containing selection operators, and that there is a complete set of transformations to obtain it. A similar form which generalizes inner and outer products is introduced with transformations appropriate to obtain it. Finally, the relation between these operators and others in APL is discussed.

This kind of analysis is important for several reasons. First, in its own right it contributes to the understanding of the operators and data-structures in APL. Second, and most important for this work, it provides a strong mathematical basis for the design of the machine to be discussed later. In particular, the ideas discussed here are reflected in the drag-along and beating processes, which are fundamental in the proposed machine design.

A. On Meta-Notation

APL is a programming language, and as such is best suited for describing processes, while mathematics is primarily concerned with discussing relations rather than processes. Thus, in order to do mathematics with APL, it is necessary to use some notations that are not available in the language itself. Some of these meta-notations are actually extensions of the language which might well be included in APL to make it more powerful, while others are necessitated by the analytic approach, and do not reflect shortcomings in APL. In the next section, definitions of objects not in APL are clearly noted as such.

B. Preliminary Definitions

The definitions to follow are given partly in APL and partly in meta-notation. Hence this and the remaining sections in this chapter assume a minimal "reading knowledge" of APL. The APL summary in Appendix A will be helpful to the reader not fluent in this language. Also recommended are the APL\360 Primer (Berry [1969]) and APL\360 Reference Manual (Pakin [1968]). At first, it might appear that defining APL operators in terms of other (intuitively but not formally defined) APL operators is elliptical. In fact, there is no circularity since the definitions could be given in more primitive forms, but at the cost of less perspicuity. Since the goal here is not the development of a coherent theory of APL expressions but rather the illumination of the behavior of these expressions, the current mode of explication was chosen. The use of "undefined" APL operators is made advisedly and no special or esoteric applications of them are made in the following definitions. The basic problem here is that of using a formalism to describe a formalism. At some point it is necessary to assume a previous knowledge of something in order to avoid an infinite regress. "Nothing can be explained to a stone; the reader must understand something beforehand." (McCarthy [1964], p. 7)

The definitions will be numbered Dn for easier reference. Theorems and transformations will be numbered Tn and TRn, respectively. In APL expressions to follow, the convention that unparenthesized subexpressions associate to the right will be used wherever this does not lead to confusion. Material which can be skipped in the first reading is enclosed in heavy brackets. For the most part, this includes formal statements in definitions which are necessary for proving theorems and correctness of transformations, but which are not essential to understanding the content of this chapter.

D0. Identity: (Meta) If $\mathscr{A}$ and $\mathscr{B}$ are expressions, then

$$\mathscr{A} \leftrightarrow \mathscr{B}$$

means they have identical values.

The sign '↔' is used for identity because the more traditional equality sign '=' is reserved for use as a dyadic scalar operator in APL.

D1. Conditional Expression: (Meta) The conditonal expression

$$\underline{IF}\ B\ \underline{THEN}\ A\ \underline{ELSE}\ C$$

has as its value the value of A if $B \leftrightarrow 1$, the value of C if $B \leftrightarrow 0$, and is undefined otherwise.

McCarthy [1963] discusses formal properties of conditional expressions, some of which are used in the proofs in this chapter.

D2. Index Origin: (Meta) The index origin is the lower bound on subscripts in APL expressions. It will be referred to as $\underline{IORG}$.

In general, this work attempts to show explicit dependencies on index origin. However, to do so throughout simply complicates many expressions without adding insight. Whenever it is unstated we use 1-origin indexing.

D3. Interval Function: If N is a non-negative integer scalar, the interval function of N, denoted by ιN, is a vector of length N whose first element is $\underline{IORG}$, and whose successive elements increase by 1.

[Formally, $\iota N \leftrightarrow \underline{IF}\ N{=}0\ \underline{THEN}\ EMPTY\ VECTOR\ \underline{ELSE}\ (\iota N\text{-}1),N{+}\underline{IORG}\text{-}1.$]

Thus, one representation for the empty vector is $\iota 0$.

D4. Odometer Function: (Meta) If R is a vector of non-negative integers, the odometer function of R, denoted by ιR, is a matrix with dimension $(\times/R),\rho R$

whose rows are the mixed-radix representation to base *R*, of the (×/ρ*R*) consecutive integers, starting with *IORG*. This extension is not a part of APL, but is useful for discussing individual subscripts of an array. [Formally, for each *I*∊⍳×/*R*, (⍳*R*)[*I*;] ↔ *IORG*+*R*⊤*I*-*IORG*.]

Example:

```
⍳3,2 ↔ 1 1
       1 2
       2 1
       2 2
       3 1
       3 2
```

D5. Row Membership: *ELT* is a function whose left operand is a vector and whose right operand is a matrix, defined as follows:

```
L ELT R ↔ IF (ρL)=(ρR)[2] THEN ∨/R∧.=L ELSE 0.
```

That is, the relation is true (has value 1) if and only if the left operand vector is identical to one of the rows in the right operand matrix.

Example:

```
(1,3) ELT ⍳3,2 ↔ 0
(2,2) ELT ⍳3,2 ↔ 1
```

D6. List:(Meta) If *L* is a vector, then the list of *L*, denoted by ;/*L*, is a subscript list made up of the elements of *L*. That is,

```
;/L ↔ L[1];L[2];...;L[ρL].
```

Example:

```
M[;/⍳5] ↔ M[1;2;3;4;5]
```

D7. Ravel: The ravel of *M*, denoted by ,*M* , is a vector containing the elements of *M* in row-major order. The dimension is

```
ρ,M ↔ ×/ρM
```

If *M* is a scalar, then ,*M* is a one-element vector.

[Otherwise for each I∊ι×/ρM,   (,M)[I] ↔ M[;/(ιρM)[I;]]]

Example:

```
,1 3
 5 7   ↔ 1,3,5,7,9,11
 9 11

,1,3,5 ↔ 1,3,5
```

D8. Reshape: Let R be a vector of non-negative integers. Then the R reshape of M , denoted by RρM, is an array with dimension R, whose elements are taken from M (possibly with repetition) in row-major order.

[Formally, for each   L ELT ιR,

(RρM)[;/L] ↔ (,M)[IORG+(×/ρM)|R⊥L-IORG]]

Example:

```
     (3,2)ρι6 ↔ 1 2
                3 4
                5 6

  4ρ1,2,3,4,5 ↔ 1,2,3,4

     (2,4)ρι3 ↔ 1 2 3 1
                2 3 1 2
```

D9. Partial Subscripting: (Meta) M[[K] S] denotes the partial subscripting of array M along the $K^{\underline{th}}$ coordinate. In other words,

```
M[[K] S] ↔ M[;...;S;...;]
              ↑    ↑    ↑
              1    K   ρρM
```

[Formally,

ρM[[K] S] ↔ ((K-1)↑ρM),(ρS),(K↓ρM)

and for each   L ELT ιρM[[K] S],

if S is a vector, then

(M[[K] S])[;/L] ↔ M[;/((K-1)↑L),S[L[K]],K↓L]

and if S is a scalar, then

(M[[K] S])[;/L] ↔ M[;/((K-1)↑L),S,(K-1)↓L]]

D10. Subscripting: If *M* is a rank-*K* array, then for any *S*1,*S*2,...,*SKM*1,*SK*

M[*S*1;...;*SKM*1;*SK*] ↔ (...((*M*[[ρρ*M*] *SK*])[[(ρρ*M*)-1] *SKM*1])...)[[1] *S*1]

The above simply gives a formal definition for array subscripting. It looks more complex than it really is because APL uses a different syntax for subscripting than for other operators. If we write *SK* *X*[*K*] *M* instead of *M*[[*K*] *S*], then the value of the above expression can be rewritten as:

*S*1 *X*[1] ... *SKM*1 *X*[(ρρ*M*)-1] *SK* *X*[ρρ*M*] *M*

D11. J-Function: Let *LEN* be a non-negative integer, *ORG* an integer, and *S*∊0,1. Then *J* *LEN*,*ORG*,*S* is an interval vector of length *LEN* whose least element is *ORG*; if *S* ↔ 0 then successive elements increase by 1, else they decrease by 1. Formally,

J *LEN*,*ORG*,*S*

↔ *IF* *S*=0 *THEN* *ORG*+(ι*LEN*)-*IORG* *ELSE* (*LEN*+*ORG*-1)-((ι*LEN*)-*IORG*).

J-vectors are a generalization of the interval function. In particular, J-vectors can have any origin, are invariant under changes of *IORG*, and can run forward or backward.

Example: *J* 4,2,0 ↔ 2,3,4,5

J 4,2,1 ↔ 5,4,3,2 and these relations are true for any *IORG*.

D12. Subarray: (Meta) Let *M* be any array and *F* an array with dimension (ρρ*M*),3. Then the subarray selected by *F*, denoted *F*∧*M*, is

F∆*M* ↔ *M*[*J* *F*[1;];*J* *F*[2;]; ... ;*J* *F*[ρρ*M*;]]

where the elements of *F* are assumed to be in the domain of the above expression.

A subarray selected by this function is <u>compact</u>. The subarray function will be used to provide a standard representation for all the various ways of selecting compact subarrays.

```
Example:   Let  ρM ↔ 10,15

           and  F̲ ↔ 4 3 0
                    3 5 1

           then  FΔM ↔ M[J̲ 4,3,0 ; J̲ 3,5,1]

                     ↔ M[3,4,5,6 ; 7,6,5]
```

D13. <u>Whole Array</u>: (Meta) For any array *M*, the whole array of *M*, denoted by Δ*M*, produces as a result the *F* such that *F*Δ*M* ↔ *M*.

[Formally, Δ*M* ↔ ⍉(3,ρρ*M*)ρ(ρ*M*),((ρρ*M*)ρ<u>*IORG*</u>), (ρρ*M*)ρ0]

```
Example:   If ρM ↔ 6,10,32,  then ΔM ↔  6 1 0
             and  IORG ↔ 1              10 1 0
                                        32 1 0
```

D14. <u>Cross Section:</u> (Meta) Let *M* be any array, *F* an array with dimension (ρρ*M*),2 such that

(i) *F*[;1]∊0,1

(ii) (~*F*[;1])/*F*[;2] ↔ (+/~*F*[;1])ρ0

(iii) (*F*[;1]/*F*[;2]) <u>*ELT*</u> ι*F*[;1]/ρ*M*

Then the *F* cross section of *M*, denoted by *F*Δ*M*, is: ρ*F*Δ̲*M* ↔ (~*F*[;1])/ρ*M* and for each *L* <u>*ELT*</u> ιρ*F*Δ̲*M*, (*F*Δ̲*M*)[;/*L*] ↔ *M*[;/(×/*F*)+(~*F*[;1])*L*]

Cross section is used to formalize the subscripting of arrays by scalars. The first column of *F* contains zeros for coordinates to be left intact. Condition (ii) require's that if *F*[*J*;1] ↔ 0 then *F*[*J*;2] ↔ 0. This is primarily to make some of the theorems easier to prove. Entries of 1 in *F*[;1] correspond to coordinates indexed by scalars in the corresponding element of *F*[;2].

Example: Let ρM ↔ 4,7,13

```
F ↔ 1 2
    0 0
    1 10
```

then FΔM ↔ M[2; ;10]

D15. **Take**: If M is any array and A is an integer vector with ρA ↔ ρρM and (|A)≤M , then A↑M is an array of the same rank of M , as follows: for each I∊ιρρM, if A[I]≥0 then include the first A[I] elements along the I-th coordinate of M; otherwise if A[I]<0 then take the last |A[I] elements.

[Formally, A↑M ↔ FΔM
where F ↔ ⍉(3,ρρM)ρ(|A),(IORG+(A<0)×(ρM)-|A),(ρρM)ρ0]

D16. **Drop**: If M and A are as above, then A↓M is similar to the take except that for each coordinate, the first (or last) |A[I] elements are ignored.

[Formally, A↓M ↔ GΔM
where G ↔ ⍉(3,ρρM)ρ((ρM)-|A),(IORGι0⌈A),(ρρM)ρ0]

D17. **Reversal**: If M is any array then ⌽[K]M is the reversal of M along the K-th coordinate.

[Formally ⌽[K]M ‹ › HΔM
where H ↔ ⍉(3,ρρM)ρ(ΔM)[;1],(ΔM)[;2],K=ιρρM]

If the subscript on the operator is elided, it is taken to be ρρM.

Example: Let

```
M ↔ 1 2 3
    4 5 6
    7 8 9
```

```
then,   (2,2)↑M ↔ 1 2                 (2,¯2)↑M ↔ 2 3
                  4 5                            5 6

        (2,1)↓M ↔ 8 9                 (¯1,1)↓M ↔ 2 3
                                                 5 6

        ⍉[1]M ↔ 7 8 9
                4 5 6
                1 2 3
```

D18. **Transpose:** If *M* is any array and *A* is an integral vector satisfying

(i) ρ*A* ↔ ρρ*M*

(ii) ∧/*A*∊ιρρ*M* i.e., *A* contains only coordinate numbers of *M*

(iii) ∧/(ι⌈/*A*)∊*A* i.e., *A* is dense

then the transpose *A*⍉*M* of *M* by *A* is defined as follows:

1. ρρ*A*⍉*M* ↔ 1+(⌈/*A*)-*IORG*

2. For each *I*∊ιρρ*A*⍉*M*,

   (ρ*A*⍉*M*)[*I*] ↔ ⌊/(*A*=*I*)/ρ*M*

3. For each *L* *ELT* ιρ *A*⍉*M*,

   (*A*⍉*M*)[;/*L*] ↔ *M*[;/*L*[*A*]]

In other words, *A* permutes the coordinates of *M*. Transpose can also specify an arbitrary diagonal slice.

Example: Suppose *M* is a matrix with ρ*M* ↔ 5,6. Then if *R* ↔ (2,1)⍉*M* , and *IORG* ↔ 1 we have ρρ*R* ↔ 1+2-1 ↔ 2 . Further, (ρ*R*)[1] ↔ ⌊/(1=2,1)/5,6 ↔ 6 (ρ*R*)[2] ↔ ⌊/(2 ↔ 2,1)/5,6 ↔ 5 and for each *L* *ELT* ι6,5, *R*[;/*L*] ↔ *M*[;/(,*L*)[2,1]] or *R*[*L*[1]; *L*[2]] ↔ *M*[*L*[2]; *L*[1]].

Thus, *R* is the ordinary matrix transpose of *M*.

Now suppose *M* is same as above and *R* ↔ (1,1)⍉*M*. Then, ρρ*R* ↔ 1+1-1 ↔ 1. So the result is a vector. Then (ρ*R*)[1] ↔ ⌊/(1=1,1)/5,6 ↔ 5.

Then for each L∊ι5, we have R[L] ↔ M[;/(,L)[1,1]]

↔ M[L ; L]

So R is the main diagonal of M.

D19. Compression: If X is any vector and U is a logical vector of the same length, then U/X is the result of suppressing from X all elements whose corresponding entry in U is 0. For an arbitrary array X, U/[I] X compresses X along the $I^{\underline{th}}$ coordinate.

[Formally, for vector X, ρU/X ↔ +/U and for each I∊ιρU,

IF U[I]=1 THEN(U/X)[+/I↑U] ↔ X[I]

This is not a constructive formula for (U/X)[I]; however, such a formula is too complex to be useful here. For any array X ,

U/[I] X ↔ X[[I] U/ι(ρX)[I]].]

D20. Expansion: If X is any vector and U is a logical vector with +/U ↔ ρX, then U\X is a vector with 0 elements wherever U has, and whose other elements are taken from X in order.

The definition of expansion is extended to higher-dimensional arrays in the same way as for compression.

[Formally, ρU\X ↔ ρU and for each I∊ιρU,

(U\X)[I] ↔ IF U[I] THEN X[+/I↑U] ELSE 0]

Example: (1,1,0,1,0)/1,2,3,4,5 ↔ 1,2,4

(1,1,0,1,0)\1,2,3 ↔ 1,2,0,3,0

C. The Standard Form for Select Expressions

In this section the selection operators considered are take, drop, reversal, transpose, and subscripting by scalars or J-vectors. Because of the similarity among the selection operators, we might expect that an expression consisting only of selection operators applied to a single array could be expressed equivalently in terms of some simpler set of operators. This expectation is fulfilled in the standard form for select expressions, to be discussed below.

If the existence of a standard form is to be at all useful, there must be a way to decide whether a particular expression has a standard form representation and if so, there must be an effective method to obtain it. In the sequel we show that every select expression has an equivalent standard form, and exhibit a set of formal transformations which are sufficient to derive the standard form from an arbitrary expression.

It may at first seem strange to include subscripting in the set of selection operators, since its parameters are of a different kind than those for the other select operators. In the other select operators such as take or drop, the left operand is a count, which is independent of ways of accessing the argument array. On the other hand, in subscripting the arguments act like maps rather than counts. For example, an expression like *A*↑*M* has meaning out of context, as long as the values of *A* and *M* are known. Contrariwise the expression *M*[1;3] cannot be evaluated without knowledgc of the index origin. In the theorems and proofs to follow, the major complications often come from this dichotomy in the way of specifying select operations, rather than from the actual content of the material. Subscripting is included because its effect is similar to the other selection operators, all of which change only the dimensions and orderings of their operands.

D21. Select Expression: Let $\mathscr{E}$ be any (well-formed) array-valued expression. Then $\mathscr{F}$ is as a select expression on $\mathscr{E}$ if it is a well-formed expression consisting of an arbitrary number (including 0) of the following operators applied to $\mathscr{E}$:

(i) Take

(ii) Drop

(iii) Reversal

(iv) Transpose

(v) Subscripting by scalars of J-vectors

By extension, we will also include the subarray and cross section operators in this class.

Example: Let *M* be a rank-3 array. Then by D21,

```
(2,1,3)⍉(⌽[2](4,¯6,3)↓M)[; ; J̲6,2,1]
```

is a select expression on *M*, but

```
-M[; ; 5,7,3,1]
```

is not because it contains the scalar operator '-' and the subscripting is not by a scalar or J-vector. The definition also admits *M* as a select expression on *M*.

D22. Equivalence Transformation: An equivalence transformation on expressions is a rule of the form:

if set of assertions then $\mathscr{E}$ => $\mathscr{F}$

where $\mathscr{E}$ and $\mathscr{F}$ are expressions. If the set of assertions is true, then expression $\mathscr{E}$ may be replaced by expression $\mathscr{F}$, and the truth of the assertions guarantees that $\mathscr{E}$=>$\mathscr{F}$.

For example (if *X* is any vector then ⌽⌽*X*=>*X*) is an equivalence transformation, since it is always true that if *X* is any vector, ⌽⌽*X* ↔ *X*.

For any given transformation, it is necessary to prove that it is indeed equivalence-preserving. If this is the case the transformation is said to be correct. Note that the notions of expression and transformation and standard form used here are informal ones. It is possible to make them rigorous, so as to be acceptable to a logician, but that is irrelevant to the current aims and would only serve to obfuscate the important mathematical relationships we are trying to explicate. The correctness proof for each transformation will be called "Proof of TRn".

D23. Standard Form: A select expression on an array *M* is in standard form (SF) if it is represented as *A*⍉*F*∆*G*⍙*M* where *A*,*F*,*G* are all of the correct size and domain.

In the remainder of this section, we introduce a set of equivalence transformations sufficient to transform most select expressions into standard form. In the process we prove the correctness of each transformation. The effect of this process is a proof of the following important theorem:

COMPLETENESS THEOREM 1: If $\mathscr{E}$ is any select expression on an array *M*, then $\mathscr{E}$ can be transformed into an equivalent expression $\mathscr{F}$ in standard form.

In order to obtain an SF representation of an arbitrary select expression, we must first be able to eliminate the operators take, drop, reversal and subscripting. The first four transformations below do this.

TR1. If *M* is any array and *A* is conformable to *M* for take, then *A*↑*M* => *F*∆*M* where *F* ↔ ⍉(3,⍴⍴*M*)⍴(|*A*),(*IORG*+(*A*<0)×(⍴*M*)-|*A*),(⍴⍴*M*)⍴0 .

TR2. If M is any array and A is conformable to M for drop, then A↓M => FΔM

where F ↔ ⍉(3,ρρM)ρ((ρM)-|A),(I̲O̲R̲G̲+0⌈A),(ρρM)ρ0.

TR3. If M is any array then ⌽[K]M => FΔM

where F ↔ ⍉(3,ρρM)ρ(ΔM)[;1],(ΔM)[;2],K=⍳ρρM.

These three transformations are obviously correct, as they follow directly from the definitions of the operators take, drop, and reversal. Their proofs will thus be omitted.

TR4. If M is any array then M[[K] J̲ LEN,ORG,S] -> FΔM

where F[K;] ↔ LEN,ORG,S and (K≠⍳ρρM)/[1]F ↔ (K≠⍳ρρM)/[1]ΔM

That the above is an equivalence transformation requires a small proof:

Proof of TR4:

We must prove that for any array M,

M[[K] J̲ LEN,ORG,S] ↔ FΔM

where F is as given in TR4. In order to prove the identity, we show first that both quantities have the same dimensions. Then we show that corresponding elements of each are identical.

Let R ↔ M[[K] J̲ LEN,ORG,S].

1. By definition, ρR ↔ ((K-1)↑ρM),(ρ J̲ LEN,ORG,S),K↓ρM

↔ ((K-1)↑ρM),LEN,K↓ρM

and ρFΔM ↔ F[;1]

↔ ((K-1)↑(ΔM)[;1]),LEN,K↓(ΔM)[;1]

↔ ((K-1)↑ρM),LEN,K↓ρM

↔ ρR

2. For each $L\ \underline{ELT}\ \iota\rho R$,

$$R[;/L] \leftrightarrow M[;/((K-1)\uparrow L),(\underline{J}\ LEN,ORG,S)[L[K]],K\downarrow L]$$

and $(F\Delta M)[;/L] \leftrightarrow (M[\underline{J}\ F[1;]\ ;\ \underline{J}\ F[2;]\ ;\ \ldots\ ;\ \underline{J}\ F[\rho\rho M;]][;/L]$

$$\leftrightarrow M[(\underline{J}\ F[1;])[L[1]];\ \ldots\ ;\ (\underline{J}\ F[\rho\rho M;])[L[M]]]$$

(by L3 in Appendix B).

But for each $I\neq K$, $(\underline{J}\ F[I;])[L[I]] \leftrightarrow (\underline{J}\ (\rho M)[I],\underline{IORG},0)[L[I]]$

$\leftrightarrow L[I]$ (by L4, Appendix B)

and $(\underline{J}\ F[K;])[L[K]] \leftrightarrow (\underline{J}\ LEN,ORG,S)[L[K]]$. Therefore,

$(F\Delta M)[;/L] \leftrightarrow M[L[1]\ ;\ L[2]\ ;\ \ldots\ ;\ L[K-1]\ ;\ (\underline{J}\ LEN,ORG,S)[L[K]];$
$L[K+1];\ \ldots\ ;L[\rho\rho M]]$

$$\leftrightarrow M[;/((K-1)\uparrow L),(\underline{J}\ LEN,ORG,S)[L[K]],K\downarrow L]$$

$$\leftrightarrow R[;/L]\quad QED.$$

The preceding proof of TR4 is reasonably simple, and is representative of the kind of proof required. Although similar in style, the proofs of the remaining transformations are more complex. Since they add little to the exposition, they are given in Appendix B.

The following transformation makes it possible to reduce the number of occurrances of adjacent subarray operators in an expression.

__TR5.__ If M is any array and F and G are conformable for subarrays, then

$$F\Delta G\Delta M \Rightarrow H\Delta M$$

where $\rho H \leftrightarrow \rho F$ and for each $I\epsilon\iota\rho\rho M$, $H[I;] \leftrightarrow L,OR,S$

where $\underline{J}\ L,OR,S \leftrightarrow (\underline{J}\ G[I;])[\underline{J}\ F[I;]]$

Transformations TR1 through TR4 are used to eliminate instances of the operators take, drop, reversal, and indexing from select expressions by transforming them into equivalent expressions involving subarray and cross section operators. TR5 shows how to coalesce two adjacent occurrances of subarray into one. The remaining transformations, TR6 through TR10 are similar in spirit and are used to permute the remaining operations into the order required by the standard form.

TR6. If M is any array and F and G are conformable, then F⍙GΔM => G'ΔF'⍙M,

where G' ↔ (~F[;1])/[1]G

and F'[;1] ↔ F[;1]

and F'[;2] ↔

F[;1]×(G[;2]+((~G[;3])×F[;2]-IORG)+(G[;3]×(G[;1]+IORG+¯1-F[;2])))

TR7. If M is any array and F and G are conformable to M for cross section,

then F⍙G⍙M => H⍙M

where H[;1] ↔ G[;1]∨(~G[;1])\F[;1]

and H[;2] ↔ G[;2]+(~G[;1])\F[;2]

TR8. If M is any array and F,A are conformable to M for subarray and transpose, respectively, then

F∧A⍉M => A⍉F[A;]ΔM.

TR9. If M is any array, Q a scalar, J∊⍳ρρA⍉M then

(A⍉M)[[J]Q] => IF 1=ρρA⍉M THEN B⍙M ELSE A'⍉B⍙M

where A' ↔ (A≠J)/A-J<A

and B[;1] ↔ J - A

and B[;2] ↔ Q×B[;1].

TR10. If M is any array and B and A are conformable for transpose, then

B⍉A⍉M => C⍉M

where C ↔ B[A].

Now that we have transformations TR1 through TR10 which are proved correct in Appendix B, we can outline a proof of Completeness Theorem 1. First note that for any array *M*, *M* ↔ (ιρρ*M*)⍉(Δ*M*)Δ(((ρρ*M*),2)ρ0)Δ̲*M*.

1. Let $\mathscr{E}$ be any select expression on *M* which satisfies the hypotheses of the theorem. Apply TR1, TR2, and TR3 to $\mathscr{E}$ enough times to eliminate all instances of the operators take, drop, and reversal. (In order to be absolutely rigorous, we would have to prove a replacement theorem which says that if in an expression $\mathscr{A}$, an occurrance of a subexpression $\mathscr{B}$ is replaced by an equivalent subexpression $\mathscr{B}'$ (i.e., $\mathscr{B} \leftrightarrow \mathscr{B}'$), then the resulting expression $\mathscr{A}'$ is equivalent to $\mathscr{A}$, only $\mathscr{A}' \leftrightarrow \mathscr{A}$. Call the result of this operation $\mathscr{E}'$. Note that $\mathscr{E}'$ contains only subscript, Δ, and ⍉ operations. Clearly $\mathscr{E}' \leftrightarrow \mathscr{E}$ because we have applied only equivalence transformations.

2. Now for each instance of an indexed quantity, substitute the equivalent expression using partial indexing, as per definition D10. Write this using the *IX* notation mentioned there and apply TR4 to eliminate all instances of J-vector subscripts and call the resulting expression $\mathscr{E}''$. It should be obvious that $\mathscr{E}''$ has the form *S*1 *θ*1 *S*2 *θ*2 ... *SN θN M*, where the S quantities are left operands for the operators *θ* and the *θ*'s are Δ, ⍉ and *IX* in arbitrary order. Finally substitute the expression (ιρρ*M*)⍉(Δ*M*)Δ(((ρρ*M*),2)ρ0)Δ̲*M* for *M*, and note that this subexpression, call it $\mathscr{S}_N$, is in standard form. Call the resulting expression $\mathscr{T}_N$, and again note that $\mathscr{T}_N \leftrightarrow \mathscr{E}$.

3. Consider the following algorithm: at each step, the input is $\mathscr{T}_K \leftrightarrow$ *S*1 *θ*1 *S*2 *θ*2 ... *SK θK* $\mathscr{S}_K$, where $\mathscr{S}_K$ is in standard form, i.e., $\mathscr{S}_K \leftrightarrow$ *AK*⍉*FK*Δ*GK*Δ̲*M* .

(a) If *K* ↔ 0 then the algorithm is terminated. Otherwise, look at the operator *θK*. Do step 1, 2, or 3 below depending on whether *θK* is ⍉, Δ or *IX* , respectively, and return to step (a).

1. θK is transpose, ⍉ . Apply TR10 to the expression $SK⍉\mathscr{S}_K \leftrightarrow SK⍉AK⍉FK\Delta GK\underline{\Delta}M$, to get the equivalent $QK⍉FK\Delta GK\underline{\Delta}M$, where $QK \leftrightarrow SK[AK]$ and call this $\mathscr{S}_{K-1}$.

2. θK is subarray, Δ . Apply transformations TR8 and TR5 to $SK\Delta\mathscr{S}_K$ to get $SK\Delta\mathscr{S}_K \leftrightarrow SK\Delta AK⍉FK\Delta GK\underline{\Delta}M \Rightarrow AK⍉SK[AK;]\Delta FK\Delta GK\underline{\Delta}M \Rightarrow AK⍉FK'\Delta GK\underline{\Delta}M$, where FK' is obtained by TR5.

3. θK is indexing by a scalar, $IX[J]$. Apply transformations TR9, TR6, and TR7 to $SK\ IX[J]\mathscr{S}_K$, getting

$$\begin{aligned} SK\ IX[J]\ AK⍉FK\Delta GK\underline{\Delta}M &\Rightarrow AK'⍉BK\underline{\Delta}FK\Delta GK\underline{\Delta}M \\ &\Rightarrow AK'⍉FK'\Delta BK'\underline{\Delta}GK\underline{\Delta}M \\ &\Rightarrow AK'⍉FK'\Delta GK'\underline{\Delta}M. \end{aligned}$$

In each of steps 1, 2, 3 above, a set of transformations was applied to the subexpression $SK\ \theta K\mathscr{S}_K$ of $\mathscr{T}_K$. Call the resulting subexpression $\mathscr{S}_{K-1}$. Since all transformations were equivalence transforms, it is clear that $SK\ \theta K\mathscr{S}_K \leftrightarrow \mathscr{S}_{K-1}$. Let $\mathscr{T}_{K-1}$ be the resulting expression from plugging $\mathscr{S}_{K-1}$ into $\mathscr{T}_K$. Clearly $\mathscr{T}_{K-1} \leftrightarrow \mathscr{T}_K$. Finally observe that each $\mathscr{S}_K$ is in standard form. Hence, in N steps, the algorithm will terminate with result $\mathscr{T}_0 \leftrightarrow \mathscr{T}_1 \leftrightarrow \dots \leftrightarrow \mathscr{T}_N \leftrightarrow \mathscr{E}$, and $\mathscr{T}_0 \leftrightarrow \mathscr{S}_0$, which is in standard form. This is the desired result. QED.

So far, we have defined a standard form for a subset of select expressions and exhibited a complete set of transformations for obtaining the standard form representation of an arbitrary expression in this class. Moreover, the proof of the completeness theorem gives an algorithm for obtaining the SF of an expression. Note that there are alternate ways of formulating the standard form. For instance, an equivalent formulation says that an expression is in standard form if it is represented as $A⍉B{\uparrow}C{\downarrow}\phi[K]\ D\underline{\Delta}M$ with B,C non-negative and K a vector of indices so that the definition of $\phi[K]$ extends in the obvious way. The choice of using the meta-notation formulations was made for two major reasons. First, fewer transformations and therefore fewer proofs are needed to establish completeness. Second, this formulation is closer to the way these results will be used in the design of the machine.

Another point to note is that the standard form could be made more general, by allowing more operators to be included in the set of selection operators. In particular, compression and expansion might be included, as well as reshape and catenation. The general rotation operator at first seems to be a possible candidate for inclusion, but in fact does not fit in cleanly. This is primarily because rotations involve taking residues of subscripts, which do not compose in a simple way. A further extension would allow arbitrary indexing of select expressions and perhaps extend operations on select expressions to operations on their subscripts, as in the case $\phi V[S] \leftrightarrow V[\phi S]$.

A final point concerns the significance of the SF and completeness results. These results are important in that they establish formally some of the relationships between APL-like operators which informally may appear obvious. This not only provides a useful tool for the programmer, who may make formal transformations on his programs without a second thought, but it also provides a formal basis for automatic transformation of programs and expressions. This second property is heavily used in the design of the APL machine. Also important is that results such as we have described aid in the understanding of array operators, which might be used in generalizing them further or in strengthening the theoretical foundation for operations on array data.

D. The Relation Between Select Operators and Reduction

Obviously there is more to APL than just selection operators. If the results of the previous section are to be generally applicable, we must look into the relationships between select operators and some of the other kinds of operators in an array language. One result that has been used implicitly in some of the proofs in Section C is that selection operators are distributive with respect to scalar arithmetic operators. For instance, $(A+B)[S] \leftrightarrow A[S]+B[S]$ and $-\phi V \leftrightarrow \phi - V$. This property follows immediately from the definition of scalar arithmetic operators and the definitions of the select operators, and is stated formally in the theorem T1 below:

<u>T1</u>. Let A and B be arrays with the same dimensions and $\underline{M}$ and $\underline{D}$ be monadic and dyadic scalar arithmetic operators and $\underline{T}$ a selection operator; then

(i) if $A\ \underline{D}\ B$ is defined,

$$\underline{T}\ (A\ \underline{D}\ B) \leftrightarrow (\underline{T}\ A)\ \underline{D}\ (\underline{T}\ B)$$

(ii) if $\underline{M}\ A$ is defined

$$\underline{T}\ \underline{M}\ A \leftrightarrow \underline{M}\ \underline{T}\ A$$

T1 contains the restriction that $A\ \underline{D}\ B$ and $\underline{M}\ A$ be defined, in order to deal with cases like $((1,1,1)+1,1,0)[1,2]$ in which the result is undefined as written but is defined after distributing the indexing operator. This result is in fact more general than as stated. It should be clear that the operator $\underline{T}$ can also be rotation, compression, expansion (for some scalar operators) or operators such as ravel or reshape. A similar result holds if one of A or B is a scalar.

One of the most important constructions in APL is reduction which applies a dyadic scalar operator between all elements of a vector. Reduction is not an operator in the sense we have been using, but is more like a functional. As will be shown below, it is possible to change the order of select operators and reductions as well as to permute the coordinates of the reducee. As in the previous section, these facts will have direct use in the APL machine. The remainder of this section defines reduction formally, and presents a set of equivalence transformations for expressions involving reductions.

D24. Reduction: If $\underline{D}$ is a dyadic scalar operator and V is a vector, then the $\underline{D}$ reduction of V, written $\underline{D}/V$, is a scalar defined as follows:

$$\underline{D}/V \leftrightarrow \underline{IF}\ (\rho V)>1\ \underline{THEN}\ V[1]\ \underline{D}\ V[2]\ \underline{D}\ \ldots\ \underline{D}\ V[\rho\rho V]$$

$$\underline{ELSE}\ \underline{IF}\ (\rho V) = 1\ \underline{THEN}\ V[1]\ \underline{ELSE}\ (IDENTITY\ OF\ \underline{D})$$

In the expression above, the operators $\underline{D}$ associate to the right, as usual. The identities of the scalar dyadic operators are listed in Appendix C. If M is any array and $\underline{D}$ is as above then the $\underline{D}$ reduction over the K th coordinate of M is defined as follows:

$$\rho\underline{D}/[K]\ M \leftrightarrow ((K-1)\uparrow\rho M),K\downarrow\rho M$$

and for each $L\ \underline{ELT}\ \iota\rho\underline{D}/[K]\ M$

$$(\underline{D}/[K]\ M)[;/L] \leftrightarrow \underline{D}/F\underline{\Delta}M$$

where $F[;1] \leftrightarrow K\neq\iota\rho\rho M\ AND\ F[;2] \leftrightarrow F[;1]\backslash L$

If the subscript K is elided in the expression $\underline{D}/[K]\ M$, it is taken to be the last coordinate of M, which is $\rho\rho M$ in 1-origin and $\lceil/\iota\rho\rho M$ in general.

In order to do some of the proofs required by this section, we will need to use the membership and ranking operators, so these operators are defined formally first.

D25. Membership: If A is a scalar and B is any array, then the membership relation $A\in B$ has value 1 if at least one of the elements of B is identical to A, otherwise the value is 0. The dimension of the result is the same as that of A, and the definition is extended element-by-element on A.

[That is $A\in B \leftrightarrow \underbrace{\vee/\ \ldots\ \vee/}_{\rho\rho B\ TIMES}A\circ.{=}B.$]

D26. Ranking: If B is a vector and A is a scalar, then $B\iota A$ denotes the index of A in B, namely the least subscript I of B such that $A \leftrightarrow B[I]$.

[Formally, $B\iota A \leftrightarrow \lfloor/(A{=}B,A)/\ \iota 1{+}\rho B.$]

From the expression above, it is clear that if ~*A*∊*B* then the result is 1+⌈/ιρ*B*. The operation is extended to arbitrary arrays *A* element-by-element.

[Thus, if *A* is any array, then for each *L* E̲L̲T̲ ιρ*A*,

(*B*ι*A*)[;/*L*] ↔ ⌊/(*A*[;/*L*] = *B*,*A*[;/*L*])/ι1+ρ*B*.]

An interesting question about reductions is under what circumstances can the coordinates of the reducee be permuted, with reduction carried out on a different coordinate, and still have the result remain the same? It is intuitively obvious, for example, that +/⌈1⌉ *M* ↔ +/⌈2⌉ (2.1)⍉*M*. when *M* is a matrix, since adding the rows is the same as adding the columns of the transpose. Theorem T2 shows that this kind of permuting can be carried out as long as the coordinates that are left after reduction are in the same order.

T2. Let *M* be any array, D̲ any scalar dyadic operator, *K* a scalar, and *P* any permutation of ιρρ*M*. Then,

D̲/[*K*] *M* ↔ D̲/[*P*[*K*]] *P*⍉*M*

if and only if

(*P*[*K*]≠ιρρ*M*)/*P*ιιρ*P* ↔ (*K*≠ιρρ*M*)/ιρρ*M*

Proof: See Appendix B.

The complicated condition in T2 is a formal statement of the requirement that permutation by *P* does not disturb the ordering of the coordinates in *M* other than *K*.

Example: Let *M* be a rank-4 array. Then, by theorem T2, all of the following are true:

```
+/[2]M ↔ +/[1] (2,1,3,4)⍉M
       ↔ +/[3] (1,3,2,4)⍉M
       ↔ +/[4] (1,4,2,3)⍉M
```

No other values of *P* satisfy the condition in T2. For instance if *P* ↔ 4,2,1,3, *P*[2] ↔ 2 and *P*ιιρ*P* ↔ 3,2,4,1. So (2≠1,2,3,4)/3,2,4,1 ↔ 3,4,1 which is not (2≠1,2,3,4)/1,2,3,4 ↔ 1,3,4. This theorem suggests the following transformation:

TR11. If *M* is any array and $\underline{D}$ is a dyadic scalar operator, then

$\underline{D}$/[*K*] *M* ↔ $\underline{D}$/[*LAST*] *A*⍉*M*.

where *LAST* is the index of the last coordinate of *M* (ρρ*M* for 1-origin and ⌈/ιρρ*M* in general) and *A* ↔ (ι*K*-1),*LAST*,((*K*-1)+ι(ρρ*M*)-*K*)

TR11 above and TR12, TR13, and TR14 to follow can be used to transform a select expression on a reduction to a reduction along the last coordinate of a select expression.

TR12. If *M* is any array and $\underline{D}$ a dyadic scalar operator then

A⍉$\underline{D}$/*M* => $\underline{D}$/(*A*,1+⌈/*A*)⍉*M*.

TR13. If *M* is any array, $\underline{D}$ a dyadic scalar operator, then

G∆$\underline{D}$/*M* => $\underline{D}$/*G*'∆*M*

where *G*' ↔ (ρ∆*M*)ρ(,*G*),(¯1↑ρ*M*),$\underline{IORG}$,0.

TR14. If *M* is any array, $\underline{D}$ a dyadic scalar operator, and *Q* a scalar,

then (($\underline{D}$/*M*)[[*J*]*Q*] => $\underline{D}$/*M*[[*J*]*Q*].

Proofs of TR11, TR13, TR14: Immediate from theorems T2, T3, T4.

Proof of TR12: See Appendix B.

Transformation TR11 forces all reductions to be along the last coordinate of their operand array. TR12, TR13, and TR14 permit reduction to be "factored out" of select expressions.

Given these transformations, we can extend the completeness result of the previous section as follows:

COMPLETENESS THEOREM 2: If $\mathscr{E}$ is an expression on an array M containing only selection operators and reductions, then it can be transformed into an equivalent expression $\mathscr{F}$ of the form $\underline{D}_1/\underline{D}_2/\ldots\underline{D}_K/\mathscr{F}'$ where the $\underline{D}_I$ are the reduction operators in the order they appeared in $\mathscr{E}$ and where $\mathscr{F}'$ is in standard form.

Since the proof of this theorem is similar to that for the first completeness theorem, it will be omitted. Such a proof depends on the correctness of transformations TR11 through TR14, which follow from the theorems below:

T3. If M is any array, $\underline{D}$ a dyadic scalar operator then

$$G\Delta\underline{D}/[K]M \leftrightarrow \underline{D}/[K]C'\Delta M$$

where (K≠⍳⍴⍴M)/[1]G' ↔ G AND G'[K;] ↔ (ΔM)⌊K;]

Proof: See Appendix B.

T4. For any array M and D a dyadic scalar operator,

$$G\underline{\Delta}\underline{D}/M \leftrightarrow \underline{D}/G'\underline{\Delta}M$$

where G' ↔ ((⍴⍴M),2)⍴(,G),0,0

Proof: See Appendix B.

The following example takes an expression and derives the standard form of Completeness Theorem 2.

Example: Let ⍴M ↔ 6,10,12,19 and consider the select expression with reductions:

$\mathscr{E}$ ↔ (2,1)⍉+/[1](3,7,¯4)↑×/[4]M

In each step, we note the transformations applied.

```
1.  ℰ↔(2,1)⍉+/[3](3,1,2)⍉FΔ×/[4]M                  (TR11, TR1)

        where   F ↔  3  1  0
                     7  1  0
                     4  9  0

2.  ℰ↔ +/[3](2,1,3)⍉(3,1,2)⍉×/[4]GΔM               (TR12, TR13)

        where   G ↔  3  1  0
                     7  1  0
                     4  9  0
                    19  1  0

3.  ℰ↔ +/[3](3,2,1)⍉×/[4]GΔM                        (TR10)

4.  ℰ↔ +/[3]×/[4](3,2,1,4)⍉GΔM                      (TR12)

5.  ℰ↔ +/[3]×/[4](3,2,1,4)⍉GΔH⍙M

        where   H ↔  0 0                  by definition of ⍙
                     0 0
                     0 0
                     0 0
```

The above expression is in SF.

E. The General Dyadic Form — A Generalization of Inner and Outer Products

In APL there are three ways of applying dyadic scalar operators to a pair of operands. The simplest, the scalar product, is the element-by-element application of a scalar operator to corresponding elements of conformable arrays. The next simplest is the outer product, in which the result is obtained by applying the operator to all possible pairs of elements, one from each operand array, in a specified order. Finally, the inner product is a generalization of ordinary matrix product in linear algebra, except that arbitrary (conformable) arrays may participate as operands and any pair of operators may be used. Before proceeding, let us present the formal definitions of inner and outer products.

D27. Outer Product: If M and N are arbitrary arrays and D̲ is any dyadic scalar operator, then the D̲ outer product of M and N, written M ∘.D̲ N, is dcfined as follows: ρM ∘.D̲ N ↔ (ρM),ρN. Then for each L E̲L̲T̲ ιρM ∘.D̲ N, (M ∘.D̲ N)[;/L] ↔ M[;/(ρρM)↑L] D̲ N[;/(ρρM)↓L].

D28. Inner Product: If M and N are any arrays such that ¯1↑ρM ↔ 1↑ρN and if D̲ and F̲ are two dyadic scalar operators, then theD̲-F̲ inner product of M and N written M D̲.F̲ N, is defined as follows: ρM D̲.F̲ N ↔ (¯1↓ρM),1↓ρN and for each L E̲L̲T̲ ιρM D̲.F̲ N, (M D̲.F̲ N)[;/L] ↔ D̲/(G⍙M) F̲ H⍙N,

where G[;1] ↔ ((¯1+ρρM)ρ1),0   G[;2] ↔ ((¯1+ρρM)↑L),0

H[;1] ↔ 0,(¯1+ρρN)ρ1

H[;2] ↔ 0,(1-ρρN)↑L

If one of M or N is a scalar, it is extended to a vector of the same length as the reduction coordinate. In the sequel, we assume that all operands of inner product are array-shaped (or have already been extended).

Example:

```
(1,2,3) ∘.× 4,5 ↔  4  5
                   8 10
                  12 15

(1,2,3) ⌈.+ 4,5,6 ↔ ⌈/(1,2,3)+4,5,6

                  ↔ 9
```

If M and N are conformable matrices, then

M +.× N

is the ordinary matrix product of linear algebra.

Although these three product forms appear to be different syntactically and also in their effect, they are in fact intimately related, and can be considered as aspects of the same thing. This section shows the close relationship between scalar, inner, and outer products, and introduces a new (meta) form which includes these as special cases. We also investigate the effect of select operations on this new construction called the general dyadic form (GDF), and show that it, like the standard form on select expressions, is closed under application of select operations.

The key to the relationship between these apparently diverse constructions is the generalized transpose operation. By applying a transpose to an outer product, it is possible to write an expression which specifies a diagonal slice of the original outer product. For example, if *V* is a vector, *M* a matrix, then the expression 1 1 2⍉*V*∘.+*M* describes the result of adding *V* to each of the columns of *M*. It would be desirable to understand this expression to mean the result it describes, namely the result of adding the vector *V* to the columns of *M*, rather than the process, that is the transpose of the outer product of *V* and *M*. The difference is important for two reasons. Using the first interpretation in a situation where the expression must actually be evaluated, as in a program, requires only the pertinent elements of the result to be computed. This is especially important when the operands are large arrays. Second, some information is lost by ignoring the partial results. For example, the expression ((1,2)÷(1,0))[1] is undefined in the literal sense but the apparent intended interpretation gives the value 1. Both in the case of select expressions and in transposes of outer products this is a serious problem, as it is in direct conflict with the semantics of APL. Formally, the definition of the language renders expressions such as the one just mentioned undefined, yet this is really a matter of taste and style. My contention is that at worst this kind of situation should be an ambiguous one, since it is essentially an instance of a side effect. That is, the programmer writing such an expression should not depend on the processor of his program to indicate that a domain error occurred in the evaluation of an irrelevant partial result. If that is what he wants, there are direct ways of expressing it, such as writing A←(1,2)÷(1,0), followed by A[1]. In any case, I have taken the view that what should be evaluated is the intent of an expression, if this is perceivable, rather than the literal expression itself. Except in cases which produce side effects, both approaches compute identical values.

Theorems T5 and T6 which follow, establish the essential connections among the product forms and the transpose.

<u>T5</u>. If A and B are conformable for scalar product, and if D̲ is a dyadic scalar operator then A D̲ B ↔ ((ιρρA),ιρρB)⍉A ∘.D̲ B.

<u>Proof</u>: See Appendix B.

<u>T6</u>. If M and N are two arrays conformable for inner product and D̲ and F̲ are dyadic scalar operators, then M D̲.F̲ N ↔ D̲/A⍉M ∘.F̲ N,

where A ↔ (ι¯1+ρρM),(2ρ LAST1),(¯1+ρρM)+ι¯1+ρρN

and LAST1 is the index of second-to-last coordinates in M ∘.F̲ N

(in 1-origin this is (ρρM)+(ρρN)-1 and ⌈/ι(ρρM)+(ρρN)-1 in general).

<u>Proof</u>: See Appendix B.

<u>Example</u>: (T6) If A and B are matrices then

A +.× B ↔ +/(1,3,3,2)⍉A ∘.× B.

We can see this as follows:

```
(+/(1,3,3,2)⍉A ∘.× B)[I;J]

          ↔ +/((1,3,3,2)⍉A ∘.× B)[I;J;]

          ↔ +/A[I;]×B[;J]

          ↔ (A +.× B)[I;J]
```

In previous sections we have looked into the effect of select operators on single arrays and scalar products. A natural question then is, what is the effect of the select operators on inner and outer products. In order to approach an answer, it was necessary to discover an alternate formulation of these constructions, which facilitates this kind of analysis. Such an alternative is the general dyadic form, defined below.

D29. General Dyadic Form: An expression on two array operands R and S, with dyadic scalar operator $\underline{D}$ is in general dyadic form (GDF) if it is expressed in the form:

$$A⍉R'\ \circ.\underline{D}\ S'$$

and the following conditions are satisfied;

(i) R' and S' are the standard forms of select expressions on R and S.

(ii) A is a conformable transpose vector for which each of $(\rho\rho R')↑A$ and $(\rho\rho R')↓A$ are in ascending order, and each contains no duplicate values.

(iii) $(\rho A⍉R'\circ.\underline{D}\ S')[A] \leftrightarrow (\rho R'),\ \rho S'$

The last condition guarantees that if A takes a diagonal slice of the outer product $R'\ \circ.\underline{D}\ S'$, then the length of corresponding coordinates in R' and S' are the same. This can always be done by performing a take operation affecting these coordinates (see TR17).

Example: If V is a vector, M and N matrices, then the following are in GDF:

$$(1,1,2)⍉V\ \circ.\underline{D}\ M,$$

$$(1,3,2,3)⍉M\ \circ.\underline{D}\ (2,1)⍉N,$$

$$(1,1)⍉((1,1)⍉M)\ \circ.\underline{D}\ V$$

but the following are not in GDF because the conditions on A are not satisfied:

(1,3,3,2)⍉M ∘.$\underline{D}$ N

(1,1,1)⍉M ∘.$\underline{D}$ V

From definitions D27, D29 and Theorem T5, it is clear that the scalar product and outer product of R and S by $\underline{D}$ are special cases of the GDF, obtained by taking A ↔ (⍳⍴⍴R),⍳⍴⍴S and A ↔ ⍳(⍴⍴R)+⍴⍴S, respectively; D28 and T6 indicate that an inner product can be expressed as a reduction of a GDF.

In discussing the effect of select operators on GDF's, we will present a series of transformations, with proofs of their correctness in Appendix B. In the following transformations, let

F ↔ (⍴⍴R')↑A and G ↔ (⍴⍴R')↓A.

TR15. If W ↔ A⍉R' ∘.$\underline{D}$ S' is in GDF then HΔW => A⍉U ∘.$\underline{D}$ V where

U is the SF of R'' ↔ H[F;]ΔR'

V is the SF of S'' ↔ H[G;]ΔS'

TR16. If W is as above and Q is a scalar, then W[[J]Q] => B⍉U ∘.$\underline{D}$ V

where B ↔ (J≠A)/A-J<A and

U is the SF of $\underline{\text{IF}}$ J∊F $\underline{\text{THEN}}$ R'[[F⍳J] Q] $\underline{\text{ELSE}}$ R'

V is the SF of $\underline{\text{IF}}$ J∊G $\underline{\text{THEN}}$ S'[[G⍳J] Q] $\underline{\text{ELSE}}$ S'

TR17. If W is as above then B⍉W => (F',G')⍉U ∘.$\underline{D}$ V

where F' ↔ (M∊B[F])/M

G' ↔ (M∊B[G])/M M ↔ ⍳(⌈/B)+1-$\underline{\text{IORG}}$

U is the SF of R'' ↔ (F'⍳B[F])⍉(⍴B⍉W)[B[F]]↑R'

V is the SF of S'' ↔ (G'⍳B[G])⍉(⍴B⍉W)[B[G]]↑S'

TR18. If *M* and *N* are conformable for inner product and $\underline{D}$ and $\underline{F}$ are dyadic scalar operators, then *M* $\underline{D}$.$\underline{F}$ *N* => $\underline{D}$/*A*⍉*M*' ∘.$\underline{F}$ *N*'

where *A* ↔ (⍳¯1+⍴⍴*M*), *LAST*1,(¯1+⍴⍴*M*)+⍳⍴⍴*N*

M' is the SF of *M*

N' is the SF of (*LASTN*,⍳¯1+⍴⍴*N*)⍉*N*

*LAST*1 is the index of the second-to-last coordinate of *M* ∘.$\underline{F}$ *N*.

((⍴⍴*M*)+(⍴⍴*N*)-1 in 1-origin; ⌈/⍳(⍴⍴*M*)+(⍴⍴*N*)-1 in general)

LASTN is the index of the last coordinate of *N*.

(⍴⍴*N* in 1-origin; ⌈/⍳⍴⍴*N* in general).

These transformations are sufficient to establish:

COMPLETENESS THEOREM 3: Let $\mathscr{E}$ be an expression consisting only of reductions and select operators applied to a scalar product, inner product, or outer product of expressions $\mathscr{A}$ and $\mathscr{B}$, where $\mathscr{A}$ and $\mathscr{B}$ are select expressions on arrays *A* and *B* respectively. Then $\mathscr{E}$ can be transformed into an equivalent expression $\mathscr{F}$ of the form $\underline{D}_1/\underline{D}_2/\ldots\underline{D}_K/\mathscr{F}'$, where $\mathscr{F}'$ is in GDF and the $\underline{D}_I$'s are the reduction operators appearing in $\mathscr{E}$, in the same order. If the original expression $\mathscr{E}$ contained an inner product, $\underline{D}_K$ is the first operator of the inner product.

Proof: Similar to Completeness Theorem 1.

F. Conclusion

This chapter has discussed some of the formal mathematical properties of the operators found in APL. Of particular interest are the completeness theorems, which give conditions under which a subset of APL expressions can be put into standard form. The general idea of the standard form is that sequences of selection operators on an expression can be transformed into a shorter sequence of operations on the same expression. In other words, if $\mathscr{E}$ is an expression and $\underline{S}1, \ldots, \underline{S}K$ are selection operators, then there is a process for finding A, F, and G such that

$$\underline{S}1\ \ \underline{S}2\ \ \ldots\ \ \underline{S}K\mathscr{E} \leftrightarrow A\text{⍉}F\Delta G\underline{\Delta}\mathscr{E}.$$

Completeness Theorem 3 further shows that, in essence, selection operations on inner, outer, or scalar products can be absorbed into the individual operands. Also by Completeness Theorems 2 and 3, reductions can be factored out of select expressions.

Clearly, the whole story has not been told at this point; indeed, the contents of this chapter barely scratch the surface of the general problem of analysis of APL semantics. Even so, the results discussed are a sufficient base for the design of the APL machine discussed in the next chapters. In particular, the analysis here provides a formal basis for the beating and drag-along processes, which are the two foundations upon which the APL machine design rests.

APPENDIX A

SUMMARY OF APL

Monadic form *f B*		f	Dyadic form *AfB*	
Definition or example	Name		Name	Definition or example
+*B* ↔ 0+*B*	Plus	+	Plus	2+3.2 ↔ 5.2
-*B* ↔ 0-*B*	Negative	-	Minus	2-3.2 ↔ ¯1.2
×*B* ↔ (*B*>0)-(*B*<0)	Signum	×	Times	2×3.2 ↔ 6.4
÷*B* ↔ 1÷*B*	Reciprocal	÷	Divide	2÷3.2 ↔ 0.625
B | ⌈*B* | ⌊*B* 3.14 | 4 | 3 ¯3.14 | ¯3 | ¯4	Ceiling	⌈	Maximum	3⌈7 ↔ 7
	Floor	⌊	Minimum	3⌊7 ↔ 3
B* ↔ (2.71828..)B*	Exponential	*	Power	2*3 ↔ 8
⍟**N* ↔ *N* ↔ *⍟*N*	Natural logarithm	⍟	Logarithm	*A*⍟*B* ↔ Log *B* base *A* *A*⍟*B* ↔ (⍟*B*)÷⍟*A*
|¯3.14 ↔ 3.14	Magnitude	|	Residue	Case | *A*|*B* *A*≠0 | *B*-(|*A*)×⌊*B*÷|*A* *A*=0,*B*≥0 | *B* *A*=0,*B*<0 | Domain error
!0 ↔ 1 !*B* ↔ *B*×!*B*-1 or !*B* ↔ Gamma(*B*+1)	Factorial	!	Binomial coefficient	*A*!*B* ↔ (!*B*)÷(!*A*)×!*B*-*A* 2!5 ↔ 10 3!5 ↔ 10
?*B* ↔ Random choice from ⍳*B*	Roll	?	Deal	A Mixed Function (See Table 3.8)
○*B* ↔ *B*×3.14159...	Pi times	○	Circular	See Table at left
~1 ↔ 0 ~0 ↔1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	Relations Result is 1 if the relation holds, 0 if it does not: 3≤7 ↔ 1 7≤3 ↔ 0
		≤	Not greater	
		=	Equal	
		≥	Not less	
		>	Greater	
		≠	Not Equal	

A	*B*	*A*∧*B*	*A*∨*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

(-*A*)○*B*	*A*	*A*○*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

Primitive Scalar Functions

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 ABCD E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ 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 P⍳3 ↔ 2 5 1 2 5 in V, or 1+⍴V P⍳E ↔ 3 5 4 5 4 4⍳4 ↔ 1 5 5 5 5
Take	V↑A	Take or drop |V[I] first 2 3↑X ↔ ABC (V[I]≥0) or last (V[I]<0) EFG elements of coordinate I ¯2↑P ↔ 5 7
Drop	V↓A	
Grade up[3,5]	⍋A	The permutation which ⍋3 5 3 2 ↔ 4 1 3 2 would order A (ascending or descending) ⍒3 5 3 2 ↔ 2 1 3 4
Grade down[3,5]	⍒A	
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 ⌽P ↔ 7 5 3 2 ⌽[1]X ↔ ⊖X ↔ IJKL EFGH ABCD
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 E∊P ↔ 0 1 1 0 1 0 1 0 0 0 0 0 P∊⍳4 ↔ 1 1 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?S	W?Y ↔ Random deal of W elements from ⍳Y

Primitive Mixed Functions

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 $E ↔ \begin{matrix}1&2&3&4\\5&6&7&8\\9&10&11&12\end{matrix}$ $X ↔ \begin{matrix}ABCD\\EFGH\\IJKL\end{matrix}$

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.

Type of Array	ρA	ρρA	ρρρA
Scalar		0	1
Vector	N	1	1
Matrix	M N	2	1
3-Dimensional	L M N	3	1

Dimension and Rank Vectors

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]

Inner Products for Primitive Scalar Dyadic Functions f and g

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]

Outer Products for Primitive Scalar Dyadic Function g

Case	ρR	Definition
R←1⍉V	ρV	R←V
R←1 2⍉M	ρM	R←M
R←2 1⍉M	(ρM)[2 1]	R[I;J]←M[J;I]
R←1 1⍉M	⌊/ρM	R[I]←M[I;I]
R←1 2 3⍉T	ρT	R←T
R←1 3 2⍉T	(ρT)[1 3 2]	R[I;J;K]←T[I;K;J]
R←2 3 1⍉T	(ρT)[3 1 2]	R[I;J;K]←T[J;K;I]
R←3 1 2⍉T	(ρT)[2 3 1]	R[I;J;K]←T[K;I;J]
R←1 1 2⍉T	(⌊/(ρT)[1 2]),(ρT)[3]	R[I;J]←T[I;I;J]
R←1 2 1⍉T	(⌊/(ρT)[1 3]),(ρT)[2]	R[I;J]←T[I;J;I]
R←2 1 1⍉T	(⌊/(ρT)[2 3]),(ρT)[1]	R[I;J]←T[J;I;I]
R←1 1 1⍉T	⌊/ρT	R[I]←T[I;I;I]

Transposition

APPENDIX B

This appendix contains proofs for the transformations and theorems which were deferred from the main part of Chapter II. They were omitted from the text because they do not substantially contribute to the exposition of the material, and are included here for completeness.

The various proofs are trying to establish the identity of two expressions $\mathscr{E}$ and $\mathscr{F}$. This is generally done in two steps: in step 1, $\rho\mathscr{E} \leftrightarrow \rho\mathscr{F}$ is shown and in step 2, it is shown that the expressions are identical element-by-element.

Lemmas L1 through L9 state results used in the rest of this appendix. Since they are all intuitively obvious, and since their proofs follow from the definitions, these proofs will be omitted.

L1. If *M* is any array and *V* is a vector, then

(M[[K] V])[[K] U] ↔ M[[K] V[U]]

L2. If *M* is any array, *I*<*J* , and *U* and *V* are vectors or scalars, then

(M[[J] V])[[I] U] ↔ (M[[I] U])[[J-0=ρρU] V]

L3. Let *M* be any array and *S1*,*S2*,...,*SK* be subscript vectors. Then for each *L* $\underline{ELT}$ιρM[S1;S2;...;SK],

(M[S1;S2;...;SK])[;/L] ↔ M[;/T]

where *T* is a vector with T[I] ↔ SI[L[I]]

for each I∊ιρρM.

L4. For any integral *A* (scalar or array) satisfying A≥$\underline{IORG}$ and (A-$\underline{IORG}$)<LEN,

a. ($\underline{J}$ LEN,ORG,0)[A] ↔ ORG+A-$\underline{IORG}$

b. ($\underline{J}$ LEN,ORG,1)[A] ↔ ORG+LEN+$\underline{IORG}$+¯1-A

c. $(\underline{J}\ LEN,ORG,S)[A] \leftrightarrow ORG+((\sim S)\times(A-\underline{IORG}))+(S\times(LEN+\underline{IORG}+{}^{-}1-A))$

d. $-\underline{J}\ LEN,ORG,S \leftrightarrow \underline{J}\ LEN,(-(ORG+LEN-1)),\sim S$

e. $K+\underline{J}\ LEN,ORG,S \leftrightarrow \underline{J}\ LEN,(ORG+K),S$ if K is an integer

f. $\phi\underline{J}\ LEN,ORG,S \leftrightarrow \underline{J}\ LEN,ORG,\sim S$

L5. If $F\Delta M$ is defined, then

(a) $\rho F\Delta M \leftrightarrow F[;1]$

(b) for each L $\underline{ELT}$ $\iota\rho F\Delta M$,

$(F\Delta M)[;/L] \leftrightarrow M[;/F[;2]+((\sim F[;3])\times(L-\underline{IORG}))+(F[;3]\times(F[;1]+\underline{IORG}+{}^{-}1-L))]$

L6. a. $U/X[S] \leftrightarrow X[U/S]$

b. $U\backslash U/X \leftrightarrow U\times X$ (if X is numeric)

c. $U/U\backslash X \leftrightarrow X$

d. $U/V/X \leftrightarrow (V\backslash U)/X$

e. $(U\wedge V)/X \leftrightarrow (U/V)/(U/X)$

f. $U/(X\ \underline{D}\ Y) \leftrightarrow (U/X)\ \underline{D}\ (U/Y)$ for $\underline{D}$ a dyadic scalar operator

g. If $\underline{D}$ is a dyadic scalar operator with $0\ \underline{D}\ 0 \leftrightarrow 0$,

then $U\backslash(X\ \underline{D}\ Y) \leftrightarrow (U\backslash X)\ \underline{D}\ (U\backslash Y)$

L7. If $0\le ORG1-\underline{IORG}$ and $(ORG1+LEN1-\underline{IORG})<LEN$ then

a. $(\underline{J}\ LEN,ORG,0)[\underline{J}\ LEN1,ORG1,S] \leftrightarrow \underline{J}\ LEN1,(ORG+ORG1-\underline{IORG}),S$

b. $(\underline{J}\ LEN,ORG,1)[\underline{J}\ LEN1,ORG1,S] \leftrightarrow \underline{J}\ LEN1,(ORG+LEN+\underline{IORG}-(ORG1+LEN1)),\sim S.$

L8. If U and V are logical vectors with $\rho V \leftrightarrow +/\sim U$

then $\sim(U\vee(\sim U)\backslash V) \leftrightarrow (\sim U)\backslash\sim V.$

L9. a. If B is a vector and if for any A, $A\epsilon B$ is all ones, then $B[B\iota A] \leftrightarrow A$.

b. If P is a permutation of $\iota\rho P$ then if $R \leftrightarrow P\iota\iota\rho P$, $P[R] \leftrightarrow R[P] \leftrightarrow \iota\rho P$ and $P \leftrightarrow R\iota\iota\rho R$. In other words, for permutation vectors, the ranking operator is its own inverse.

Proof of TR5:

1. $\rho F\Delta G\Delta M \leftrightarrow \rho F[;1] \leftrightarrow \rho H\Delta M$ (by L5)

2. For each $L\ \underline{ELT}\ \iota\rho F\Delta G\Delta M$, $(F\Delta G\Delta M)[;/L] \leftrightarrow (G\Delta M)[;/S]$

   where $S[I] \leftrightarrow (\underline{J}\ F[I;])[L[I]]$

   and $(G\Delta M)[;/S] \leftrightarrow M[;/T]$

   where $T[I] \leftrightarrow (\underline{J}\ G[I;])[S[I]]$

   $\leftrightarrow (\underline{J}\ G[I;])[(\underline{J}\ F[I;])[L[I]]]$

   $\leftrightarrow ((\underline{J}\ G[I;])[\underline{J}\ F[I;]])[L[I]]$

   But $(H\Delta M)[;/L] \leftrightarrow M[;/U]$

   where $U[I] \leftrightarrow (\underline{J}\ H[I;])[L[I]]$

   $\leftrightarrow ((\underline{J}\ G[I;])[\underline{J}\ F[I;]])[L[I]]$

   $\leftrightarrow T[I]$

   Thus, $T \leftrightarrow U$ and $(F\Delta G\Delta M)[;/L] \leftrightarrow (H\Delta M)[;/L]$. QED.

We can give explicit formulas for H in TR5. First, $H[;1] \leftrightarrow F[;1]$ and $H[;3] \leftrightarrow F[;3]\neq G[;3]$. Finally, for each $I \epsilon\, \iota\rho\rho M$, $H[I;2] \leftrightarrow \underline{IF}\ 0{=}G[I;3]$ $\underline{THEN}\ F[I;3]{+}G[I;3]{-}\underline{IORG}\ \underline{ELSE}\ (\underline{IORG}{+}{+}/G[I;1,2]){-}{+}/F[I;1,2]$.

Proof of TR6:

1. $\rho F\underline{\Delta}G\Delta M \leftrightarrow (\sim F[;1])/\rho G\Delta M$

   $\leftrightarrow (\sim F[;1])/G[;1]$

   $\leftrightarrow G'[;1] \leftrightarrow \rho G'\wedge F'\underline{\Delta}M$.

2. For oaoh $L\ \underline{ELT}\ \iota\rho F\underline{\Delta}G\wedge M$,

   $(F\underline{\Delta}G\Delta M)[:/L] \leftrightarrow (G\Delta M)[;/L']$ where $L' \leftrightarrow (\times/F){+}(\sim F[;1])\backslash L$ (by D14)

   $\leftrightarrow M[;/S]$

where (by L5),

$$S \leftrightarrow G[;2]+((\sim G[;3])\times L'-\underline{IORG})+(G[;3]\times(G[;1]+\underline{IORG}+{}^{-}1-L')$$

$$\leftrightarrow G[;2]+((\sim G[;3])\times(\times/F)+((\sim F[;1])\backslash L)-\underline{IORG})$$

$$+(G[;3]\times(G[;1]+\underline{IORG}+{}^{-}1-((\times/F)+(\sim F[;1])\backslash L))$$

$$(G'\Delta F'\underline{\Delta}M)[;/L] \leftrightarrow (F'\underline{\Delta}M)[;/T]$$

where $T \leftrightarrow G'[;2]+((\sim G'[;3])\times L-\underline{IORG})+(G'[;3]\times(G'[;1]+\underline{IORG}+{}^{-}1-L))$

Thus, $(G'\Delta F'\underline{\Delta}M)[;/L] \leftrightarrow M[;/U]$

where $U \leftrightarrow (\times/F')+(\sim F'[;1])\backslash T$

$$\leftrightarrow (\times/F')+(\sim F'[;1])\backslash(G'[;2]+((\sim G'[;3])\times L-\underline{IORG})$$

$$+(G'[;3]\times(G'[;1]+\underline{IORG}+{}^{-}1-L)))$$

To complete the proof, we need to show that $S \leftrightarrow U$. By lemma L6g,

$$X\backslash A+B \leftrightarrow (X\backslash A)+(X\backslash B),$$

and $X\backslash A\times B \leftrightarrow (X\backslash A)\times(X\backslash B)$.

Thus, writing $E \leftrightarrow \sim F'[;1] \leftrightarrow \sim F[;1]$, and substituting for F',

$$U \leftrightarrow (F[;1]\times(F[;1]\times G[;2]+((\sim G[;3])\times F[;2]-\underline{IORG})$$

$$+(G[;3]\times(G[;1]+\underline{IORG}+{}^{-}1-F[;2]))))$$

$$+(E\backslash G'[;2])+((E\backslash\sim G'[;3])\times(E\backslash L)-\underline{IORG})$$

$$+(E\backslash G'[;3])\times(E\backslash G'[;1])+\underline{IORG}+{}^{-}1-E\backslash L$$

But $E\backslash G'[;K] \leftrightarrow E\times G[;K] \leftrightarrow (\sim F[;1])\times G[;K]$ for $K\in 1,2,3$.

Making this substitution and commuting terms,

$$U \leftrightarrow ((F[;1]+\sim F[;1])\times(G[;2]+((\sim G[;3])\times-\underline{IORG})+G[;3]\times G[;1]+\underline{IORG}-1)$$

$$+((\sim G[;3])\times(F[;1]\times F[;2])+(\sim F[;1])\times(\sim F[;1])\backslash L)$$

$$+G[;3]\times(F[;1]\times-F[;2])+(\sim F[;1])\times-(\sim F[;1])\backslash L\ .$$

But $F[;1]+\sim F[;1] \leftrightarrow (\rho F[;1])\rho 1$ and does not contribute to the product in the first term. Also,

$$(\sim F[;1])\times(\sim F[;1])\backslash L \leftrightarrow (\sim F[;1])\backslash L.$$

$$U \leftrightarrow G[;2]+((\sim G[;3])\times(\times/F)+((\sim F[;1])\backslash L)+\underline{IORG})$$

$$+G[;3]\times G[;1]+\underline{IORG}+{}^{-}1-((\times/F)+(\sim F[;1])\backslash L)$$

$\leftrightarrow S$ **QED.**

<u>Proof of TR7</u>:

1. $\rho F\underline{\Delta}G\underline{\Delta}M \leftrightarrow (\sim F[;1])/\rho G\underline{\Delta}M \leftrightarrow (\sim F[;1])/(\sim G[;1])/\rho M$

$$\leftrightarrow ((\sim G[;1])\backslash\sim F[;1])/\rho M \qquad \textbf{(by L6d)}$$

$$\rho H\underline{\Delta}M \leftrightarrow (\sim H[;1])/\rho M \leftrightarrow (\sim(G[;1]\vee(\sim G[;1])\backslash F[;1]))/\rho M$$

$$\leftrightarrow ((\sim G[;1])\backslash\sim F[;1])/\rho M \qquad \textbf{(by L8)}$$

$$\leftrightarrow \rho F\underline{\Delta}G\underline{\Delta}M$$

2. For each L $\underline{ELT}$ $\iota\rho F\underline{\Delta}G\underline{\Delta}M$,

$$(F\underline{\Delta}G\underline{\Delta}M)[;/L] \leftrightarrow (G\underline{\Delta}M)[;/(\times/F)+(\sim F[;1])\backslash L] \leftrightarrow M[;/S]$$

where $S \leftrightarrow (\times/G)+(\sim G[;1])\backslash(\times/F)+(\sim F[;1])\backslash L$

$$(H\underline{\Delta}M)[;/L] \leftrightarrow M[;/(\times/H)+(\sim H[;1])\backslash L] \leftrightarrow M[;/T]$$

where $T \leftrightarrow ((G[;1]\vee(\sim G[;1])\backslash F[;1])\times(G[;2]+(\sim G[;1])\backslash F[;2]))$

$$+(\sim(G[;1)\vee(\sim G[;1])\backslash F[;1]))\backslash L$$

Expanding the products, and noting that

$$G[;1]\vee(\sim G[;1])\backslash F[;1] \leftrightarrow G[;1]+(\sim G[;1])\backslash F[;1],$$

we get

$$T \leftrightarrow (\times/C)+(G[;1]\times(\sim G[;1])\backslash F[;2])+(G[;2]\times(\sim G[;1])\backslash F[;1])$$

$$+(((\sim G[;1])\backslash F[;1])\times(\sim G[;1])\backslash F[;2])+((\sim G[;1])\backslash\sim F[;1])\backslash L.$$

So we must show that $S \leftrightarrow T$. In simplifying T, we use the following, in order: If U and V are logical vectors,

$$U\times(\sim U)\backslash X \leftrightarrow (\rho U)\rho 0$$

$$(U\backslash X)\times(U\backslash Y) \leftrightarrow U\backslash X\times Y \qquad \textbf{(L6g)}$$

$$U\backslash V\backslash X \leftrightarrow (U\backslash V)\backslash X$$

Also recall from the definition of $\underline{\Delta}$ that $G[;2]$ contains zeros wherever $G[;1]$ does. Thus, we rewrite T:

$$T \leftrightarrow (\times/G)+(G[;2]\times(\sim G[;1])\backslash F[;1])+((\sim G[;1])\backslash(\times/F))+((\sim G[;1])\backslash\sim F[;1])\backslash L$$

But the second term goes away because of $G[;2]$'s zeros.

$$T \leftrightarrow (\times/G)+((\sim G[;1])\backslash(\times/F))+(\sim G[;1])\backslash(\sim F[;1])\backslash L$$

$$\leftrightarrow (\times/G)+(\sim G[;1])\backslash((\times/F)+(\sim F[;1])\backslash L)$$

$$\leftrightarrow S \text{ QED.}$$

Proof of TR8:

Clearly the ranks of both expressions are identical.

1. $\rho F\Delta A⍉M \leftrightarrow F[;1]$ **(by L5)**

   Now, for each $I\epsilon\iota\rho\rho A⍉F[A;]\Delta M$

   $$(\rho A⍉F[A;]\Delta M)[I] \leftrightarrow \lfloor/(A=I)/\rho F[A;]\Delta M \leftrightarrow \lfloor/(A=I)/F[A;1]$$

   $$\leftrightarrow \lfloor/F[(A=I)/A;1] \qquad \textbf{(by L6a)}$$

   $$\leftrightarrow \lfloor/(+/A=I)\rho F[I;1] \leftrightarrow F[I;1] \leftrightarrow (\rho F\Delta A⍉M)[I]$$

2. For each L $\underline{ELT}$ $\iota\rho F\Delta A⍉M$,

   $$(F\Delta A⍉M)[;/L] \leftrightarrow (A⍉M)[;/Q] \leftrightarrow M[;/Q[A]]$$

   where $$Q[I] \leftrightarrow (\underline{J}\ F[I;])[L[I]]$$

   $$(A⍉F[A;]\Delta M)[;/L] \leftrightarrow (F[A;]\Delta M)[;/L[A]] \leftrightarrow M[;/S]$$

   where $$S[I] \leftrightarrow (\underline{J}\ (F[A;])[I;])[(L[A])[I]]$$

   $$\leftrightarrow (\underline{J}\ F[A[I];])[L[A[I]]]$$

   $$\leftrightarrow Q[A[I]] \leftrightarrow (Q[A])[I]$$

   Thus $Q[A] \leftrightarrow S$. QED.

Proof of TR9: The case of (ρA⍉M) ↔ 1 is trivial and will be omitted. Otherwise,

1. ρρ(A⍉M)[[J]Q] ↔ (ρρA⍉M)-1 ↔ (⌈/A)-1   (in 1-origin)

ρρA'⍉B⍙M ↔ ⌈/A' ↔ ⌈/(A≠J)/A-J<A ↔ ⌈/((A≠J)/A-A<J)[L,E,G]   (*)

where L,E,G exhausts ιρA and such that ∧/A[L]<J and

∧/A[E]=J and ∧/A[G]>J . (This is possible by commutativity of ⌈.)

(*) ↔ ⌈/(J≠A[L,E,G])/A[L,E,G]-J<A[L,E,G]

↔ ⌈/(((ρA[L])ρ1),((ρA[E])ρ0),(ρA[G])ρ1)/(A[L],A[E],A[G])

((ρA[L,E])ρ0),(ρA[G])ρ1

↔ ⌈/A[L],(A[G]-1) ↔ (⌈/A[L])⌈(⌈/A⌊G])-1

If J ↔ ⌈/A then A[G] ↔ ι0 and the result is ⌈/A[L] ↔ (⌈/A)-1. Otherwise, A[G] is non-empty and ⌈/A[G] ↔ ⌈/A, so the result is still (⌈/A)-1 , since A exhausts ιρA, by definition. Thus the ranks of both expressions are identical. We now show the dimensions to be indentical.

For each I∊ι(⌈/A)-1,

(ρA'⍉B⍙M)[I] ↔ ⌊/(I=A')/ρB⍙M ↔ ⌊/(I=(A≠J)/A-J<A)/(A≠J)/ρM

↔ ⌊/((A≠J)/I=A-J<A)/(A≠J)/ρM ↔ ⌊/((A≠J)∧I=A-J<A)/ρM (by L6e)

By case analysis, we find that

(A≠J)∧I=A-J<A ↔ <u>IF</u> I<J <u>THEN</u> I=A <u>ELSE</u> (I+1)=A

↔ A=I+I≥J

Thus, (ρA'⍉B⍙M)[I] ↔ ⌊/(A=I+I≥J)/ρM ↔ (ρA⍉M)⌊I+I≥J]   (by D18)

and (ρ(A⍉M)[[J]Q])[I] ↔ ((J≠ιρA)/ρA⍉M)[I]

↔ (ρA⍉M)[((J≠ιρA)/ιρρA⍉M)[I]]

↔ (ρA⍉M)[I+I≥J] ↔ (ρA'⍉B⍙M)[I]

Therefore both expressions have the same dimension.

2. For each $L\ \underline{ELT}\ \iota\rho(A⍉M)[[J]Q]$,

$$((A⍉M)[[J]Q])[;/L] \leftrightarrow (A⍉M)[;/((J-1)\uparrow L),Q,(J-1)\downarrow L]$$

$$\leftrightarrow M[;/(((J-1)\uparrow L),Q,(J-1)\downarrow L)[A]]$$

Call this subscript vector S.

$$(A'⍉B\underline{\Delta}M)[;/L] \leftrightarrow (B\underline{\Delta}M)[;/L[A']] \leftrightarrow M[;/(\times/B)+(\sim B[;1])\backslash L[A']]$$

Call this subscript vector T. It remains to show that $S \leftrightarrow T$. First, $\rho S \leftrightarrow \rho T$. For each $I\in\iota\rho S$,

$$S[I] \leftrightarrow (((J-1)\uparrow L),Q,(J-1)\downarrow L)[A[I]]$$

$$\leftrightarrow \underline{IF}\ A[I]<J\ \underline{THEN}\ L[A[I]]\ \underline{ELSE}\ \underline{IF}\ A[I]=J\ \underline{THEN}\ Q\ \underline{ELSE}\ ((J-1)\downarrow L)[A[I-J]]$$

So, $S \leftrightarrow (Q\times J=A)+(J\neq A)\times L[A-J<A]$.

$$T \leftrightarrow (Q\times J=A)+(J\neq A)\backslash L[(A\neq J)/A-J<A] \leftrightarrow (Q\times J=A)+(J\neq A)\backslash(J\neq A)/L[A-J<A]$$

$$\leftrightarrow (Q\times J=A)+(J\neq A)\times L[A-J<A]$$

$\leftrightarrow S$ QED.

<u>Proof of TR10:</u> As in the proof of TR9, the hard part of this proof is to show that the two expressions $B⍉A⍉M$ and $B[A]⍉M$ have the same dimension.

1. Clearly $B[A]⍉M$ is well-defined since A exhausts $\iota\rho B$ and $\rho B[A] \leftrightarrow \rho\rho M$. Also, $\rho\rho B[A]⍉M \leftrightarrow \lceil/B[A] \leftrightarrow \lceil/B \leftrightarrow \rho\rho B⍉A⍉M$. By definition of transpose, for each $I\in\iota\rho\rho B⍉A⍉M$,

$$(\rho B⍉A⍉M)[I] \leftrightarrow \lfloor/(I=B)/\rho A⍉M \leftrightarrow \lfloor/(\rho A⍉M)[(I=B)/\iota\rho\rho A⍉M].$$

Let us write $R \leftrightarrow A⍉M$ and $T \leftrightarrow (I=B)/\iota\rho\rho R$. The remainder of this part depends primarily on the associativity and commutativity of minimum $(\lfloor)$.

$$(\rho B⍉A⍉M)[I] \leftrightarrow \lfloor/(\rho R)[T] \leftrightarrow \lfloor/(\rho R)[T[1]],\ (\rho R[T[2]],\ldots,(\rho R)[T[\rho\rho T]]$$

$$\leftrightarrow \lfloor/(\lfloor/(A=T[1])/\rho M),(\lfloor/(A=T[2])/\rho M),\ldots,(\lfloor/(A=T[\rho\rho T])/\rho M)$$

$$\leftrightarrow \lfloor/((A=T[1])/\rho M),((A=T[2])/\rho M),\ldots,((A=T[\rho\rho T])/\rho M)$$

$$\leftrightarrow \lfloor/((A=T[1])\vee(A=T[2])\vee\ldots\vee(A=T[\rho\rho T]))/\rho M$$

$$\leftrightarrow \lfloor/(A\in T)/\rho M \qquad \text{(by D25)}$$

Now $I{=}B[A] \leftrightarrow (I{=}B)[A]$ since I is scalar. Also note that $((I{=}B)[A])[K] \leftrightarrow 1$ if and only if $A[K]\epsilon T$. Thus, $I{=}B[A] \leftrightarrow A\epsilon T$ and

$$(\rho B[A]⍉M)[I] \leftrightarrow \lfloor/(I{=}B[A])/\rho M$$

$$\leftrightarrow \lfloor/(A\epsilon T)/\rho M \leftrightarrow (\rho B⍉A⍉M)[I].$$

2. For each $L\ \underline{ELT}\ \iota\rho B⍉A⍉M$,

$$(B⍉A⍉M)[;/L] \leftrightarrow (A⍉M)[;/L[B]]$$

$$\leftrightarrow M[;/(L[B])[A]]$$

$$\leftrightarrow M[;/L[B[A]]]$$

$$\leftrightarrow (B[A]⍉M)[;/L]$$

QED.

<u>Proof of Theorem T2</u>:

The <u>only if</u> part is easiest, as it depends only on the dimensions of the expressions involved. <u>Only if</u> part:

By hypothesis, $\underline{D}/[K]\ M \leftrightarrow \underline{D}/[P[K]]\ P⍉M$.

Thus, the dimensions of both expressions are identical. Specifically,

$$\rho\underline{D}/[K]\ M \leftrightarrow ((K-1)\uparrow\rho M),\quad K\downarrow\rho M \leftrightarrow (K\neq\iota\rho\rho M)/\rho M$$

and $\rho\underline{D}/[P[K]]\ M \leftrightarrow (P[K]\neq\iota\rho\rho P⍉M)/\rho P⍉M$

But, since P is a permutation of $\iota\rho\rho M$ then $\rho P \leftrightarrow \rho\rho M$

and $\rho P⍉M \leftrightarrow (\rho M)[P\iota\iota\rho\rho M] \leftrightarrow (\rho M)[P\iota\iota\rho P]$

Also, $\rho\rho P⍉M \leftrightarrow \rho\rho M$. Hence,

$$\rho\underline{D}/[P[K]]\ M \leftrightarrow (P[K]\neq\iota\rho\rho M)/(\rho M)[P\iota\iota\rho P]$$

$$\leftrightarrow (\rho M)[(P[K]\neq\iota\rho\rho M)/P\iota\iota\rho P] \quad (\star) \qquad \text{(by L6a)}$$

and $$\rho\underline{D}/[K]M \leftrightarrow (\rho M)[(K\neq\iota\rho\rho M)/\iota\rho\rho M] \quad (\star\star)$$

But $(\star) \leftrightarrow (\star\star)$ by hypotheses. Thus, the subscripts of (ρM) are indentical for each expression, i.e.,

$$(P[K]\neq\iota\rho\rho M)/P\iota\iota\rho P \leftrightarrow (K\neq\iota\rho\rho M)/\iota\rho\rho M.$$

We now proceed with the difficult part of the proof:

<u>If part</u>:

1. We must show that $\rho\underline{D}/[K]\ M \leftrightarrow \rho\underline{D}/[P[K]]\ P\text{⍉}M$,

   $\rho\underline{D}/[K]\ M \leftrightarrow ((K-1)\uparrow\rho M),\ K\downarrow\rho M \leftrightarrow (K\neq\iota\rho\rho M)/\rho M \leftrightarrow (\rho M)[(K\neq\iota\rho\rho M)/\iota\rho\rho M]$

   But $\rho\rho P\text{⍉}M \leftrightarrow \lceil/P \leftrightarrow \rho\rho M$. So for each $I\in\iota\rho\rho M$,

   $(\rho P\text{⍉}M)[I] \leftrightarrow \lfloor/(P=I)/\rho M \leftrightarrow \lfloor/(\rho M)[(P=I)/\iota\rho\rho M] \leftrightarrow (\rho M)[(P=I)/\iota\rho\rho M]$

   since P has exactly one element equal to I.

   $\leftrightarrow (\rho M)[P\iota I]$ (by D26)

   Hence, $\rho P\text{⍉}M \leftrightarrow (\rho M)[P\iota\iota\rho P]$. Now,

   $\rho\underline{D}/[P[K]]\ P\text{⍉}M \leftrightarrow (P[K]\neq\iota\rho\rho P\text{⍉}M)/\rho P\text{⍉}M \leftrightarrow (P[K]\neq\iota\rho\rho M)/(\rho M)[P\iota\iota\rho P]$

   $\leftrightarrow (\rho M)[(P[K]\neq\iota\rho\rho M)/P\iota\iota\rho P] \leftrightarrow (\rho M)[(K\neq\iota\rho\rho M)/\iota\rho\rho M]$

   by hypothesis

   $\leftrightarrow \rho\underline{D}/[K]\ M.$

   Thus, the dimensions are identical.

2. The two expressions are identical element-by-element.

   For each $L\ \underline{ELT}\ \iota\rho\underline{D}/[K]\ M,\quad (\underline{D}/[K]\ M)[;/L] \leftrightarrow \underline{D}/F\underline{\Delta}M$

   where $F[;1] \leftrightarrow K\neq\iota\rho\rho M$

   and $F[;2] \leftrightarrow F[;1]\backslash L$

   $(\underline{D}/[P[K]]\ P\text{⍉}M)[;/L] \leftrightarrow \underline{D}/G\underline{\Delta}P\text{⍉}M$

   where $G[;1] \leftrightarrow P[K]\neq\iota\rho\rho M$

   and $G[;2] \leftrightarrow G[;1]\backslash L$

   Let us examine these two reducees element-by-element. First note that they have the same rank. For, $\rho F\underline{\Delta}M \leftrightarrow (K=\iota\rho\rho M)/\rho M \leftrightarrow (\rho M)[K]$

   and $\rho G\underline{\Delta}P\text{⍉}M \leftrightarrow (P[K]=\iota\rho\rho M)/\rho P\text{⍉}M$

   $\leftrightarrow (\rho P\text{⍉}M)[P[K]]$

   $\leftrightarrow \lfloor/(P[K]=P)/\rho M$

   $\leftrightarrow (\rho M)[K].$

For each $I\epsilon\iota(\rho M)[K]$,

$$(F\underline{\Delta}M)[I] \leftrightarrow M[;/R]$$

where
$$\begin{aligned}R &\leftrightarrow (\times/F)+(\sim F[;1])\backslash I\\
&\leftrightarrow ((K\neq\iota\rho\rho M)\backslash L)+(K=\iota\rho\rho M)\backslash I\\
&\leftrightarrow (L,I)[(\iota K\text{-}1),(\rho\rho M),(K\text{-}1)+\iota(\rho\rho M)\text{-}K]\end{aligned}$$

$$\begin{aligned}(G\underline{\Delta}P\text{⍉}M)[I] &\leftrightarrow (P\text{⍉}M)[;/(\times/G)+(\sim G[;1])\backslash I]\\
&\leftrightarrow (P\text{⍉}M)[;/((P[K]\neq\iota\rho\rho M)\backslash L)+(P[K]=\iota\rho\rho M)\backslash I]\\
&\leftrightarrow M[;/S]\end{aligned}$$

where
$$\begin{aligned}S &\leftrightarrow ((L,I)[(\iota P[K]\text{-}1),(1+\rho L),(P[K]\text{-}1)+\iota(\rho L)\text{-}(P[K]\text{-}1)])[P]\\
&\quad ((L,I)[(\iota P[K]\text{-}1),(\rho\rho M),(P[K]\text{-}1)+\iota(\rho\rho M)\text{-}P[K]])[P]\end{aligned}$$

To complete the proof, we must show that $R \leftrightarrow S$.

In order to look more closely at S, we must find out more about P. Let

$$T \leftrightarrow P\iota\iota\rho P.$$

Then by hypothesis,

$$(P[K]\neq\iota\rho\rho M)/T \leftrightarrow (K\neq\iota\rho\rho M)/\iota\rho\rho M \leftrightarrow (\iota K\text{-}1),K+\iota(\rho\rho M)\text{-}K.$$

Since P is a permutation, $\wedge/(\iota\rho P)\epsilon P$ and we would expect to have $\wedge/(\iota\rho T)\epsilon T$. The above equation gives all of T except for the element which equals K. There are ρT places in T that K could occur, falling into three cases. By examining each of these cases, we can deduce the structure of P, and thus the value of S.

(a) $P[K] \leftrightarrow K$. Then $T \leftrightarrow (\iota K\text{-}1),K,K+\iota(\rho\rho M)\text{-}K \leftrightarrow \iota\rho\rho M$.

Thus, $P \leftrightarrow \iota\rho\rho M$ and $S \leftrightarrow R$.

(b) $P[K]<K$. Then, $T \leftrightarrow (\iota P[K]\text{-}1),K,((P[K]\text{-}1)+\iota(K\text{-}1)\text{-}(P[K]\text{-}1)),K+\iota(\rho\rho M)\text{-}K$

and by lemma L9

$$P \leftrightarrow T\iota\iota\rho T$$

$$\leftrightarrow (\iota P[K]-1),(1+(P[K]-1)+\iota(K-P[K])),P[K],K+\iota(\rho\rho M)-K$$

$$\leftrightarrow (\iota P[K]-1),(P[K]+\iota K-P[K]),P[K],K+\iota(\rho\rho M)-K$$

and then

$$S \leftrightarrow (L,I)[(\iota P[K]-1),((P[K]-1)+\iota K-P[K]),(\rho\rho M),K+\iota(\rho\rho M)-K]$$

$$\leftrightarrow (L,I)[(\iota K-1),(\rho\rho M),K+\iota(\rho\rho M)-K] \leftrightarrow R$$

(c) $P[K]>K$. In this case, $T \leftrightarrow (\iota K-1),(K+\iota P[K]-K),K,P[K]+\iota(\rho\rho M)-P[K]$

and $P \leftrightarrow T\iota\iota\rho T \leftrightarrow (\iota K-1),P[K],((K-1)+\iota P[K]-K),P[K]+\iota(\rho\rho M)-P[K]$.

Then, $S \leftrightarrow (L,I)[(\iota K-1),(\rho\rho M),((K-1)+\iota P[K]-K),(P[K]-1)+\iota(\rho\rho M)-P[K]]$

$$\leftrightarrow (L,I)[(\iota K-1),(\rho\rho M),(K-1)+\iota(\rho\rho M)-K] \leftrightarrow R.$$

Hence, in all cases $S \leftrightarrow R$ and therefore $F\underline{\Delta}M \leftrightarrow G\underline{\Delta}P\text{⍉}M$

for each $L\ \underline{ELT}\ \iota\rho\underline{D}/[K]\ M$,

and thus $\underline{D}/[K]\ M \leftrightarrow \underline{D}/[P[K]]\ P\text{⍉}M$. **QED.**

Proof of TR12:

1. The ranks of both expressions are clearly equal. Then, for each $I\in\iota\rho\rho A\text{⍉}\underline{D}/M$,

$$(\rho A\text{⍉}\underline{D}/M)[I] \leftrightarrow \lfloor/(A=I)/\rho\underline{D}/M \leftrightarrow \lfloor/(A=I)/{}^{-}1\downarrow\rho M$$

But also, for each $I\in\iota\rho\rho(A,1+\lceil/A)\text{⍉}M$,

$$(\rho(A,1+\lceil/A)\text{⍉}M)\lfloor I\rfloor \leftrightarrow \lfloor/(I=A,1+\lceil/A)/\rho M \leftrightarrow \lfloor/((I=A)/{}^{-}1\downarrow\rho M),(I=1+\lceil/A)/{}^{-}1\uparrow\rho M$$

$$SO\ \rho\underline{D}/(A,1+\lceil/A)\text{⍉}M \leftrightarrow {}^{-}1\downarrow\rho(A,1+\lceil/A)\text{⍉}M \leftrightarrow \rho A\text{⍉}\underline{D}/M$$

2. For each $L\ \underline{ELT}\ \iota\rho A\text{⍉}\underline{D}/M$,

$$(A\text{⍉}\underline{D}/M)[;/L] \leftrightarrow (\underline{D}/M)[;/L[A]] \leftrightarrow \underline{D}/F\underline{\Delta}M$$

where $F[;1] \leftrightarrow (\lceil/\iota\rho\rho M)\neq\iota\rho\rho M \leftrightarrow (({}^{-}1+\rho\rho M)\rho 1),0$

and $F[;2] \leftrightarrow F[;1]\backslash L[A] \leftrightarrow L[A],0$

$$(\underline{D}/(A,1+\lceil/A)\text{⍉}M)[;/L] \leftrightarrow \underline{D}/G\underline{\Delta}(A,1+\lceil/A)\text{⍉}M$$

where $G[;1] \leftrightarrow (\lceil/\iota\rho\rho(A,1+\lceil/A)\text{⍉}M)\neq\iota\rho\rho(A,1+\lceil/A)\text{⍉}M$

$$\leftrightarrow (({}^{-}1+\rho\rho(A,1+\lceil/A)\text{⍉}M)\rho 1),0$$

$$\leftrightarrow ((\rho\rho A\text{⍉}\underline{D}/M)\rho 1),0$$

$$G[;2] \leftrightarrow G[;1]\backslash L \leftrightarrow L,0$$

A typical element of this reducee is

$$(G\underline{\Delta}(A,1+\lceil/A)\text{⍉}M)[I] \leftrightarrow ((A,1+\lceil/A)\text{⍉}M)[;/(\times/G)+(\sim G[;1])\backslash I]$$

$$\leftrightarrow ((A,1+\lceil/A)\text{⍉}M)[;/(L,0)+((\rho L)\rho 0),I]$$

$$\leftrightarrow M[;/(L,I)[A,1+\lceil/A] \leftrightarrow M[;/L[A],I] \leftrightarrow (F\underline{\Delta}M)[I]$$

Thus, the two reducees are equal. QED.

Proof of Theorem T3:

1. $\rho G\Delta\underline{D}/[K]\ M \leftrightarrow G[;1]$

$$\rho\underline{D}/[K]\ G'\Delta M \leftrightarrow (K\neq\iota\rho\rho M)/\rho G'\Delta M$$

$$\leftrightarrow (K\neq\iota\rho\rho M)/G'[;1] \leftrightarrow G[;1] \leftrightarrow \rho G\Delta\underline{D}/[K]\ M$$

2. For each L $\underline{ELT}$ $\iota\rho G\Delta\underline{D}/[K]\ M$,

$$(G\Delta\underline{D}/[K]\ M)[;/L] \leftrightarrow (\underline{D}/[K]\ M)[;/S] \leftrightarrow \underline{D}/F\underline{\Delta}M$$

where $S \leftrightarrow G[;2]+((\sim G[;3])\times L-\underline{IORG})+G[;3]\times G[;1]+\underline{IORG}+{}^{-}1-L$

and $F[;1] \leftrightarrow K\neq\iota\rho\rho M$

and $F[;2] \leftrightarrow F[;1]\backslash S$

$$(\underline{D}/[K]\ G'\Delta M)[;/L] \leftrightarrow \underline{D}/F'\underline{\Delta}G'\Delta M$$

where $F'[;1] \leftrightarrow K\neq\iota\rho\rho G\Delta M \leftrightarrow K\neq\iota\rho\rho M$ and $F'[;2] \leftrightarrow F'[;1]\backslash L$

But by TR6, $F'\underline{\Delta}G'\Delta M \leftrightarrow G''\Delta F''\underline{\Delta}M$

where $G'' \leftrightarrow (\sim F'[;1])/[1]G' \leftrightarrow (\Delta M)[K;]$

and $F''[;1] \leftrightarrow F'[;1] \leftrightarrow F[;1]$,

$$F''[;2] \leftrightarrow F'[;1]\times G'[;2]+((\sim G'[;3])\times F'[;2]-\underline{IORG})+G'[;3]\times G'[;1]$$
$$+\underline{IORG}+{}^{-}1-F'[;2]$$

But `F'[;1]×F'[;2] ↔ F'[;2]`

and for `J∊1,2,3.`

`F'[;1]×G'[;J] ↔ F[;1]\G[;J]`

Thus, distributing the `F'[;1]` term and substituting,

`F"[;2] ↔ (F[;1]\G[;2])+((F[;1]\(~G[;3]))×(F[;1]\L)-IORG)`

`+(F[;1]\G[;3])×(F[;1]\G[;1])+IORG+¯1-F[;1]\L`

`↔ F[;1]\G[;2]+((~G[;3])×L-IORG)+G[;3]×G[;1]+IORG+¯1-L`

`↔ F[;1]\S ↔ F[;2]`

Hence `F" ↔ F`

and `G"ΔF"ΔM ↔ G"ΔFΔM ↔ FΔM` **QED.**

Proof of Theorem T4:

1. `ρGΔD/M ↔ (~G[;1])/ρD/M ↔ (~G[;1])/¯1↓ρM`

   `ρD/G'ΔM ↔ ¯1↓ρG'ΔM ↔ ¯1↓(~G'[;1])/ρM ↔ ¯1↓((~G[;1]),1)/ρM`

   `↔ (~G[;1])/¯1↓ρM ↔ ρGΔD/M`

2. For each `L ELT ιρGΔD/M,`

   `(GΔD/M)[;L] ↔ (D/M)[;/(×/G)+(~G[;1])\L] ↔ D/FΔM`

   where `F[;1] ↔ (⌈/ιρρM)≠ιρρM`

   `F[;2] ↔ F[;1]\(×/G)+(~G[;1])\L ↔ (×/G')+ F[;1]\(~G[;1])\L`

   Further, `(D/G'ΔM)[;/L] ↔ D/F'ΔG'ΔM ↔ D/HΔM`

   where `F'[;1] ↔ (⌈/ιρρG'ΔM)≠ιρρG'ΔM`

   and `F'[;2] ↔ F'[;1]\L`

   and, by TR7, `H[;1] ↔ G'[;1]∨(~G'[;1])\F'[;1]`

   `H[;2] ↔ G'[;2]+(~G'[;1])\F'[;2]`

Now for each I∊ιρρFΔ̲M,

(FΔ̲M)[I] ↔ M[;/(×/F)+(~F[;])\I]

↔ M[;/((×/G')+F[;1]\(~G[;1])\L)+(~F[;1])\I]

↔ M[;/((×/G)+(~G[;1])\L),I]

since F[;1] ↔ ((¯1+ρρM)ρ1),0

and (~G'[;1])\F'[;1] ↔ ((~G[;1]),1)\F'[;1]

↔ ((~G[;1]),1)\(¯1↓F'[;1]),¯1↑F'[;1]

↔ ((~G[;1])\(¯1+ρρG'Δ̲M)ρ1),0 ↔ (~G[;1]),0

So H[;1] ↔ G'[;1]∨(~G[;1]),0 ↔ (G[;1],0)∨(~G[;1]),0 ↔ (⌈/ιρρM)≠ιρρM

H[;2] ↔ (G[;2],0)+((~G[;1],1)\F'[;2]

↔ (G[;2],0)+((~G[;1])\¯1↓F'[;2]),0 ↔ (G[;2]+(~G[;1])\L),0

and thus (HΔ̲M)[I] ↔ M[;/(×/H)+(~H[;1])\I]

↔ M[;/(G[;2]+(~G[;1])\L),I] ↔ (FΔ̲M)[I]

and so HΔ̲M ↔ FΔ̲M.

Therefore GΔ̲D̲/M ↔ D̲/G'Δ̲M. QED.

Proof of Theorem T5: There are two main cases.

a. Onc of A or B is a scalar and is extended to the size of the other operand. Suppose A is scalar. Then, A ∘.D̲ B ↔ A D̲ B, by definition, and (ιρρA),ιρρB ↔ (ι0),ιρρB ↔ ιρρB, which is the identity transpose, and similarly if B is a scalar.

b. A and B are arrays of identical dimension. Then

1. ρρ((ιρρA),ιρρB)⍉A ∘.D̲ B ↔ (⌈/(ιρρA),ιρρB)+1-I̲O̲R̲G̲

↔ (⌈/ιρρA)+1-I̲O̲R̲G̲ ↔ ρρA

and for each I∊ιρρA,

(ρ((ιρρA),ιρρB)⍉A ∘.D̲ B)[I] ↔ ⌊/(I=(ιρρA),ιρρB)/(ρA),ρB

↔ ⌊/(I=ιρρA)/ρA ↔ (ρA)[I]

Thus, ρA D̲ B ↔ ρ((ιρρA),ιρρB)⍉A ∘.D̲ B.

2. For each $L\ \underline{ELT}\ \iota\rho A\ \underline{D}\ B$,

$$(((\iota\rho\rho A),\iota\rho\rho B)\text{⍉}A\ \circ.\underline{D}\ B)[;/L] \leftrightarrow (A\ \circ.\underline{D}\ B)[;/L,L] \leftrightarrow A[;/L]\ \underline{D}\ B[;/L]$$

$$\leftrightarrow (A\ \underline{D}\ B)[;/L]$$ QED.

Proof of Theorem T6:

1. $\rho\rho A\text{⍉}M\ \circ.\underline{F}\ N \leftrightarrow (\lceil/A)+1-\underline{IORG} \leftrightarrow \lceil/\iota(\rho\rho M)+(\rho\rho N)-1 \leftrightarrow 1+\rho\rho M\ \underline{D}.\underline{F}\ N$

For each $I\epsilon\iota\rho A\text{⍉}M\ \circ.\underline{F}\ N$,

$$(\rho A\text{⍉}M\ \circ.\underline{F}\ N)[I] \leftrightarrow \lfloor/(I=A)/\rho M\ \circ.\underline{F}\ N \leftrightarrow \lfloor/(I=A)/(\rho M),\rho N$$

$$\leftrightarrow \underline{IF}\ I\epsilon\iota\text{¯}1+\rho\rho M\ \underline{THEN}\ (\rho M)[I]\ \underline{ELSE}\ \underline{IF}\ I\epsilon(\text{¯}1+\rho\rho M)+\iota\text{¯}1+\rho\rho N$$

$$\underline{THEN}\ (\rho N)[2+I-\rho\rho M]\ \underline{ELSE}\ \lfloor/(\text{¯}1\downarrow\rho M),1\uparrow\rho N.$$

So, $\rho A\text{⍉}M\ \circ.\underline{F}\ N \leftrightarrow (\text{¯}1\downarrow\rho M),(1\downarrow\rho N),\text{¯}1\uparrow\rho M$

and therefore $\rho\underline{D}/A\text{⍉}M\ \circ.\underline{F}\ N \leftrightarrow \text{¯}1\downarrow\rho A\text{⍉}M\ \circ.\underline{F}\ N$

$$\leftrightarrow (\text{¯}1\downarrow\rho M),1\downarrow\rho N \leftrightarrow \rho M\ \underline{D}.\underline{F}\ N$$

2. For each $L\ \underline{ELT}\ \iota\rho M\ \underline{D}.\underline{F}\ N$,

$$(M\ \underline{D}.\underline{F}\ N)[;/L] \leftrightarrow \underline{D}/(G\underline{\Delta}M)\ \underline{F}\ H\underline{\Delta}N$$

where G and H are as in D28. Also, $(\underline{D}/A\text{⍉}M\ \circ.\underline{F}\ N)[;/L] \leftrightarrow \underline{D}/E\underline{\Delta}A\text{⍉}M\ \circ.\underline{F}\ N$

where $E[;1] \leftrightarrow ((\text{¯}1+\rho\rho A\text{⍉}M\ \circ.\underline{F}\ N)\rho 1),0 \leftrightarrow ((\rho\rho M\ \underline{D}.\underline{F}\ N)\rho 1),0$

and $E[;2] \leftrightarrow E[;1]\backslash L \leftrightarrow L,0$

To complete the proof, we must show that the two reducees above are identical.

Clearly both have the same dimension, namely $\text{¯}1\uparrow\rho M$.

Then for each $I\epsilon\iota\rho\text{¯}1\uparrow\rho M$,

$$((G\underline{\Delta}M)\ \underline{F}\ H\underline{\Delta}N)[I] \leftrightarrow (G\underline{\Delta}M)[I]\ \underline{F}\ (H\underline{\Delta}N)[I]$$

$$\leftrightarrow M[;/((\text{¯}1+\rho\rho M)\uparrow L),I]\ \underline{F}\ N[;/I,(-\text{¯}1+\rho\rho N)\uparrow L]$$

$$(E\underline{\Delta}A\text{⍉}M\ \circ.\underline{F}\ N)[I] \leftrightarrow (A\text{⍉}M\ \circ.\underline{F}\ N)[;/L,I] \leftrightarrow (M\ \circ.\underline{F}\ N)[;/(L,I)[A]]$$

$$\leftrightarrow (M\ \circ.\underline{F}\ N)[;/((\text{¯}1+\rho\rho M)\uparrow L),I,I,(-\text{¯}1+\rho\rho N)\uparrow L]$$

$$\leftrightarrow M[;/((\text{¯}1+\rho\rho M)\uparrow L),I]\ \underline{F}\ N[;/I,(-\text{¯}1+\rho\rho N)\uparrow L]$$

$$\leftrightarrow ((G\underline{\Delta}M)\ \underline{F}\ H\underline{\Delta}N)[I]$$

Thus, $(G\underline{\Delta}M)\ \underline{F}\ H\underline{\Delta}N \leftrightarrow E\underline{\Delta}A\text{⍉}M\ \circ.\underline{F}\ N$, and so the $\underline{D}$ reductions of each are identical. QED.

Proof of TR15:

1. The ranks of both expressions are the same since the subarray operator does not affect ranks. So for each $I \epsilon \iota\rho\rho W$,

$$(\rho A\text{⍉}U\ \circ.\underline{D}\ V)[I] \leftrightarrow \lfloor/(I=A)/\rho U\ \circ.\underline{D}\ V.$$

But $\rho U\ \circ.\underline{D}\ V \leftrightarrow (H[F;]\Delta R')\ \circ.\underline{D}\ H[G;]\Delta S'$

$\leftrightarrow (\rho H[F;]\Delta R'),\rho H[C;]\Delta S'$

$\leftrightarrow H[F;1],H[G;1] \leftrightarrow H[F,G;1] \leftrightarrow H[A;1]$

Thus, $(\rho A\text{⍉}U\ \circ.\underline{D}\ V)[I] \leftrightarrow \lfloor/(I=A)/H[A;1] \leftrightarrow \lfloor/H[(I=A)/A;1] \leftrightarrow H[I;1]$

and therefore $\rho A\text{⍉}U\ \circ.\underline{D}\ V \leftrightarrow H[;1] \leftrightarrow \rho H\Delta W$.

2. For each $L\ \underline{ELT}\ \iota\rho H\Delta W$,

$(H\Delta W)[;/L] \leftrightarrow (A\text{⍉}R'\ \circ.\underline{D}\ S')[;/P] \leftrightarrow (R'\ \circ.\underline{D}\ S')[;/P[A]]$

$\leftrightarrow R'[;/P[F]]\ \underline{D}\ S'[;/P[G]]$

where $P \leftrightarrow H[;2]+((\sim H[;3])\times L-\underline{IORG})+H[;3]\times H[;1]+\underline{IORG}+{}^{-}1-L$

$(A\text{⍉}U\ \circ.\underline{D}\ V)[;/L] \leftrightarrow (R''\ \circ.\underline{D}\ S'')[;/L[A]]$

$\leftrightarrow (H[F;]\Delta R')[;/L[F]]\ \underline{D}\ (H[G;]\Delta S')[;/L[G]]$

$\leftrightarrow R'[;/T]\ \underline{D}\ S'[;/T']$

where $T \leftrightarrow H[F;2]+((\sim H[F;3])\times L[F]-\underline{IORG})+H[F;3]\times H[F;1]+\underline{IORG}+{}^{-}1-L[F]$

$\leftrightarrow P[F]$ and similarly,

$T' \leftrightarrow P[G]$

Then $(A\text{⍉}U\ \circ.\underline{D}\ V)[;/L] \leftrightarrow R'[;/P[F]]\ \underline{D}\ S'[;/P[G]] \leftrightarrow (H\Delta W)[;/L]$.

Finally, the result is in GDF since U and V are in SF and the value of A still satisfies the required conditions. QED.

Proof of TR16:

1. $\rho W[[J]\ Q] \leftrightarrow (J \neq \iota\rho\rho W)/\rho W$. To determine $\rho B⍉U\ \circ.\underline{D}\ V$ we must first find $\rho U\ \circ.\underline{D}\ V$.

$$\rho U \leftrightarrow \rho R'' \leftrightarrow \underline{IF}\ J \in F\ \underline{THEN}\ \rho R'[[F \iota J]\ Q]\ \underline{ELSE}\ \rho R'$$

There are two cases:

a. $J \in F$. Then,

$$\rho R'' \leftrightarrow \rho R'[[F\iota J]\ Q] \leftrightarrow ((F\iota J) \neq \iota\rho\rho R')/\rho R'$$

$$\leftrightarrow ((F\iota J) \neq \iota\rho\rho R')/(\rho W)[F] \qquad \text{(by D29)}$$

$$\leftrightarrow (\rho W)[((F\iota J) \neq \iota\rho F)/F]$$

$$\leftrightarrow (\rho W)[(F \neq J)/F]$$

$$\leftrightarrow (((J-1)\uparrow\rho W),(\rho W)[J],J\downarrow\rho W)[(F \neq J)/F]$$

$$\leftrightarrow (((J-1)\uparrow\rho W),J\downarrow\rho W)[(F \neq J)/F-J<F]$$

since J does not occur in $(F \neq J)/F$

$$\leftrightarrow (\rho W[[J]\ Q])[(F \neq J)/F-J<F]$$

b. If $\sim J \in F$ then $(F \neq J) \leftrightarrow (\rho F)\rho 1$. So in this case,

$$\rho R'' \leftrightarrow \rho R' \leftrightarrow (\rho W)[F] \leftrightarrow (\rho W[[J]\ Q])[(F \neq J)/F-J<F]$$

So $\rho U \leftrightarrow (\rho W[[J]\ Q])[(F \neq J)/F-J<F]$ and similarly,

$$\rho V \leftrightarrow (\rho W[[J]\ Q])[(G \neq J)/G-J<G].$$

Therefore, $\rho U\ \circ.\underline{D}\ V \leftrightarrow (\rho W[[J]\ Q])[((F \neq J)/F-J<F),(G \neq J)/G-J<G]$

$$\leftrightarrow (\rho W[[J]\ Q])[(J \neq F,G)/(F,G)-J<F,G]$$

$$\leftrightarrow (\rho W[[J]\ Q])[(J \neq A)/A-J<A]$$

Then for each $I \in \iota\rho\rho B⍉U\ \circ.\underline{D}\ V$,

$$(\rho B⍉U\ \circ.\underline{D}\ V)[I] \leftrightarrow \lfloor/(I=B)/\rho U\ \circ.\underline{D}\ V$$

$$\leftrightarrow \lfloor/(I=(J \neq A)/A-J<A)/(\rho W[[J]\ Q])[(J \neq A)/A-J<A]$$

$$\leftrightarrow \lfloor/(\rho W[[J]\ Q])[(I=(J \neq A)/A-J<A)/(J \neq A)/A-J<A]$$

$$\leftrightarrow (\rho W[[J]\ Q])[I]$$

and thus $\rho B\text{⍉}U \circ.\underline{D}\ V \leftrightarrow \rho W[[J]\ Q]$.

2. For each $L\ \underline{ELT}\ \iota\rho W[[J]\ Q]$,

$$\begin{aligned}(W[[J]\ Q])[;/L] &\leftrightarrow W[;/((J-1)\uparrow L),Q,(J-1)\downarrow L]\\ &\leftrightarrow (R' \circ.\underline{D}\ S')[;/(((J-1)\uparrow L),Q,(J-1)\downarrow L)[A]]\\ &\leftrightarrow R'[;/T[F]]\ \underline{D}\ S'[;/T[G]]\end{aligned}$$

where $T \leftrightarrow ((J-1)\uparrow L),Q,(J-1)\downarrow L$.

$$\begin{aligned}(B\text{⍉}U \circ.\underline{D}\ V)[;/L] &\leftrightarrow (R'' \circ.\underline{D}\ S'')[;/L[B]]\\ &\leftrightarrow R''[;/(\rho\rho R'')\uparrow L[B]]\ \underline{D}\ S''[;/(\rho\rho R'')\downarrow L[B]]\end{aligned}$$

Consider the R'' term above. There are two cases, as before:

a. $\sim J\in F$. Then,

$$\begin{aligned}R''[;/(\rho\rho R'')\uparrow L[B]] &\leftrightarrow R'[;/(\rho\rho R')\uparrow L[(J\neq A)/A-J<A]]\\ &\leftrightarrow R'[;/L[(\rho\rho R')\uparrow(J\neq A)/A-J<A]]\\ &\leftrightarrow R'[;/L[(J\neq F)/F-J<F]] \leftrightarrow R'[;/L[F-J<F]]\\ &\leftrightarrow R'[;/(((J-1)\uparrow L),Q,(J-1)\downarrow L)[F]] \leftrightarrow R'[;/T[F]]\end{aligned}$$

b. $J\in F$.

$$\begin{aligned}R''[;/(\rho\rho R'')\uparrow L[B]] &\leftrightarrow (R'[[F\iota J]\ Q])[;/L[({}^{-}1+\rho\rho R')\uparrow(J\neq A)/A-J<A]]\\ &\leftrightarrow (R'[[F\iota J]\ Q])[;/L[(J\neq F)/F-J<F]]\\ &\leftrightarrow (R'[[F\iota J]\ Q])[;/L[({}^{-}1+F\iota J)\uparrow F],L[(F\iota J)\downarrow F-1]]\end{aligned}$$

because F is in ascending order and $+/J=F \leftrightarrow 1$

$$\begin{aligned}&\leftrightarrow R'[;/L[({}^{-}1+F\iota J)\uparrow F],Q,L[{}^{-}1+(F\iota J)\downarrow F]]\\ &\leftrightarrow R'[;/(((J-1)\uparrow L),Q,(J-1)\downarrow L)[(({}^{-}1+F\iota J)\uparrow F),F[J],(F\iota J)\downarrow F]]\end{aligned}$$

because of F's order

$$\leftrightarrow R'[;/T[F]]$$

And similarly, $S''[;/(\rho\rho R'')\downarrow L[B]] \leftrightarrow S'[;/T[G]]$

Thus $(W[[J]\ Q])[;/L] \leftrightarrow (B\text{⍉}U \circ.\underline{D}\ V)[;/L]$.

Finally, it is clear that the result is in GDF since U and V are in SF and B satisfies the necessary conditions. QED.

Proof of TR17:

1. $\rho\rho(F',G')⍉U \circ.\underline{D}\ V \leftrightarrow (\lceil/F',G')+1-\underline{IORG}$

$\leftrightarrow (\lceil/((M\in B[F])/M),(M\in B[G])/M)+1-\underline{IORG} \leftrightarrow (\lceil/(M\in B[F,G])/M)+1-\underline{IORG}$

$\leftrightarrow (\lceil/M)+1-\underline{IORG} \leftrightarrow (\lceil/\iota(\lceil/B)+1-\underline{IORG})+1-\underline{IORG}$

$\leftrightarrow (((\lceil/B)+1-\underline{IORG})+\underline{IORG}-1)+1-\underline{IORG} \leftrightarrow (\lceil/B)+1-\underline{IORG} \leftrightarrow \rho\rho B⍉W$

For each $I\in\iota\rho\rho B⍉W$,

$$(\rho B⍉W)[I] \leftrightarrow \lfloor/(I=B)/\rho W$$

and $(\rho(F',G')⍉U \circ.\underline{D}\ V)[I] \leftrightarrow \lfloor/(I=F',G')/\rho U \circ.\underline{D}\ V$

$\leftrightarrow \lfloor/(I=F',G')/(\rho R''),\rho S''$

So we must find $\rho R''$ and $\rho S''$.

$\rho R'' \leftrightarrow \rho(F'\iota B[F])⍉(\rho B⍉W)[B[F]]\uparrow R'$

$\rho\rho R'' \leftrightarrow (\lceil/F'\iota B[F])+1-\underline{IORG} \leftrightarrow (\lceil/\iota\rho F')+1-\underline{IORG} \leftrightarrow \rho F'$

Then, for each $J\in\iota\rho\rho R''$,

$(\rho R'')[J] \leftrightarrow \lfloor/(J=F'\iota B[F])/\rho(\rho B⍉W)[B[F]]\uparrow R'$

$\leftrightarrow \lfloor/(J=F'\iota B[F])/(\rho B⍉W)[B[F]]$

$\leftrightarrow \lfloor/(\rho B⍉W)[(J=F'\iota B[F])/B[F]]$

$\leftrightarrow \lfloor/(\rho B⍉W)[(F'[J]=B[F])/B[F]]$

$\leftrightarrow (\rho B⍉W)[F'[J]]$

Hence $\rho R'' \leftrightarrow (\rho B⍉W)[F']$

and similarly, $\rho S'' \leftrightarrow (\rho B⍉W)[G']$,

and thus $(\rho(F',G')⍉U \circ.\underline{D}\ V)[I] \leftrightarrow \lfloor/(I=F',G')/(\rho B⍉W)[F',G']$

$\leftrightarrow \lfloor/(\rho B⍉W)[(I=F',G')/F',G']$

$\leftrightarrow (\rho B⍉W)[I]$

and therefore $\rho(F',G')⍉U \circ.\underline{D}\ V \leftrightarrow \rho B⍉W$.

2. For each $L\ \underline{ELT}\ \iota\rho B⍉W$,

$$(B⍉W)[;/L] \leftrightarrow (R' \circ.\underline{D}\ S')[;/L[B[A]]]$$

$$\leftrightarrow R'[;/(\rho\rho R')\uparrow L[B[A]]]\ \underline{D}\ S'[;/(\rho\rho R')\downarrow L[B[A]]]$$

$$\leftrightarrow R'[;/L[B[F]]]\ \underline{D}\ S'[;/L[B[G]]]$$

$$((F',G')⍉U \circ.\underline{D}\ V)[;/L] \leftrightarrow (R'' \circ.\underline{D}\ S'')[;/L[F',G']]$$

$$\leftrightarrow R''[;/L[F']]\ \underline{D}\ S''[;/L[G']]$$

So we must calculate the R'' and S'' terms above.

$$R''[;/L[F']] \leftrightarrow ((F'\iota B[F])⍉(\rho B⍉W)[B[F]]\uparrow R')[;/L[F']]$$

$$\leftrightarrow ((\rho B⍉W)[B[F]]\uparrow R')[;/L[F'[F'\iota B[F]]]]$$

$$\leftrightarrow ((\rho B⍉W)[B[F]]\uparrow R')[;/L[B[F]]$$

$$\leftrightarrow R'[;/L[B[F]]]$$

since $L\ \underline{ELT}\ \iota\rho B⍉W$

implies $L[B[F]]\ \underline{ELT}\ \iota(\rho B⍉W)[B[F]]$

Similarly, $S''[;/L[G']] \leftrightarrow S'[;/L[B[G]]]$

Thus, $((F',G')⍉U \circ.\underline{D}\ V)[;/L] \leftrightarrow R'[;/L[B[F]]]\ \underline{D}\ S'[;/L[B[G]]]$

$$\leftrightarrow (B⍉W)[;/L]$$

Finally, observe that the result is in GDF since U and V are in SF and F' and G' are in order and contain no duplications by construction. QED.

<u>Proof of TR18:</u>

Immediate from T6.

APPENDIX C

IDENTITY ELEMENTS

Dyadic Function		Identity Element	Left-Right
Times	×	1	L R
Plus	+	0	L R
Divide	÷	1	R
Minus	-	0	R
Power	*	1	R
Logarithm	⍟		None
Maximum	⌈	¯7.237...E75	L R
Minimum	⌊	7.237...E75	L R
Residue	|	0	L
Circle	○		None
Out of	!	1	L
Or	∨	0	L R
And	∧	1	L R
Nor	⍱		None
Nand	⍲		None
Equal	=	1 Apply	L R
Not equal	≠	0 for	L R
Greater	>	0 logical	R
Not less	≥	1 arguments	R
Less	<	0 only	L
Not greater	≤	1	L

Identity Elements of Primitive Scalar Dyadic Functions

CHAPTER III

STEPS TOWARD A MACHINE DESIGN

Never do today what you can
Put off till tomorrow.

William Brighty Rands

procrastination is the
art of keeping
up with yesterday

Don Marquis, archy and mehitabel

As demonstrated in Chapter II, there is a high degree of power and internal consistency in the APL operators and data structures. This makes it possible to write simple expressions which have the same semantic content as several statements in comparable programming languages. This chapter discusses how to exploit these features in the design of an APL machine.

In general, APL programs contain less detail than corresponding programs in languages like ALGOL 60, FORTRAN, or PL/I. For instance, the maximum value in a vector, *V* , of data can be expressed as ⌈/*V* in APL while ALGOL requires the following:

```
MAX:=smallestnumberinmachine;
for:=1 step 1 until N do
if V[I]>MAX then MAX:=V[I]:
```

While this aspect of APL often makes programs shorter and less intricate than, say, ALGOL programs, it also requires that an evaluator of APL be more complex than one for ALGOL, especially if such expressions are to be evaluated efficiently. On the other hand, a machine doing APL has greater freedom since its behavior is specified less explicitly. In effect, APL programs can be considered as descriptions of their results rather than as recipes for obtaining them. Further, the language renders many of these descriptions obvious, both to the human reader and to a machine, as in the case of ⌈/V, while other languages encode them so intricately that the original intention of the programmer is hidden. In the example above, an APL machine can choose any method it pleases to find the maximum value while an ALGOL machine doesn't know what result is expected.

This feature of APL also has some drawbacks in that some expressions for results require unnecessary computations if calculated literally as written. For instance, the expression 3↑(2×-V) specifies a result which is the first 3 elements of twice the negative of V. Presumably the programmer is only interested in these three elements. However, the literal interpretation of this expression proceeds as follows:

1. Negate V (and store it somewhere).
2. Multiply the previous result by 2 (and store it).
3. Take the first 3 elements of the last result.

In case V is large, this process is grossly inefficient. The negation requires (ρV) fetches and stores as well as (ρV) spaces for the value to be stored. The multiplication requires another(ρV) fetches, stores, and multiplies. In fact, the desired result could have been found simply by negating the first three elements of V and multiplying by 2. Clearly, we would like the APL machine to be able to evaluate such programs efficiently!

A. Drag-Along and Beating

One approach to efficient and natural evaluation of APL expressions is to exploit the mathematical properties of the language to simplify calculations. In the machine, this approach is embodied in two fundamental new processes: drag-along and beating.

Drag-along is the process of deferring evaluation of operands and operators as long as possible. By examining a deferred expression it may be possible to simplify it in ways which are impossible when only small parts of the expression are available. In effect, drag-along makes the machine context-sensitive, while most machines are context-free.

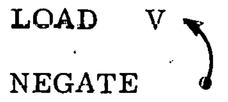
Consider the drag-along evaluation of the example in the last section. If we assume a stack machine, the machine code for this expression might be

```
1.  LOAD     V
2.  NEGATE
3.  LOAD     2
4.  MULTIPLY
5.  TAKE     3
```

The immediate execution of this sequence was already shown. Suppose now that we temporarily defer instructions in a buffer instead of executing them as they appear. After the first instruction, the buffer contains

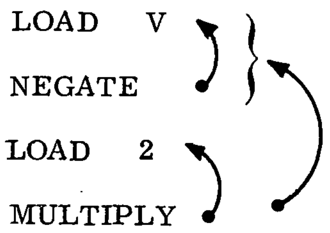
```
LOAD    V
```

After instruction 2, we have



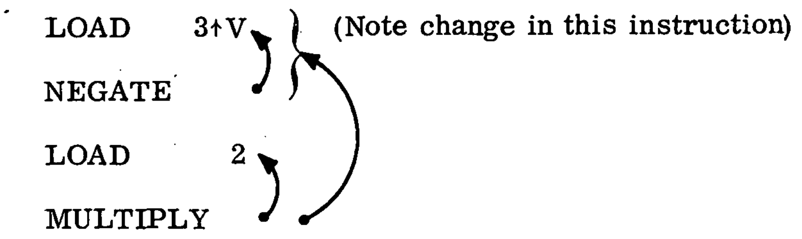
```
LOAD    V
NEGATE
```

where the pointer connects the negation with its deferred operand, V. After instruction 4, the buffer contains



```
LOAD    V
NEGATE
LOAD    2
MULTIPLY
```

The evaluation of the TAKE is different from the previous operators since it is a selection operator. TAKE can examine the contents of the buffer and change them, as below. Note that the deferred expression is equivalent to the original expression. The process of making the changes in the buffer is called beating.



```
LOAD       3↑V      (Note change in this instruction)
NEGATE
LOAD         2
MULTIPLY
```

When values must finally be computed, only the desired elements will be accessed and used. Thus, drag-along facilitates beating.

The other aspect of drag-along is that it eliminates intermediate array-shaped results with consequent savings of stores, fetches, and space. In an expression such as $A+B+C+D$ the literal execution proceeds in three steps:

$T1 \leftarrow C+D$

$T2 \leftarrow B+T1$

$T3 \leftarrow A+T2$

If the variables A,B,C,D are vectors, each step above requires a vector-sized temporary store and the last two steps require fetches to get the previous results as operands. With drag-along, the entire expression is deferred finally to be evaluated element-by-element as:

```
for I←1 step 1 until ρA do
    T3[I]←A[I]+B[I]+C[I]+D[I]
```

This requires no extra fetches, stores, or temporary space to obtain the desired result.

In the machine, drag-along will be applied to all array operands $\mathscr{E}$ and $\mathscr{F}$ and to all monadic and dyadic operators $\underline{MOP}$ and $\underline{DOP}$ for which

$$(\underline{MOP}\ \mathscr{E})[;/L] \leftrightarrow \underline{MOP'}(F1\ \mathscr{E})[;/L]$$

and

$$(\mathscr{E}\ \underline{DOP}\ \mathscr{F})[;/L] \leftrightarrow (F1\ \mathscr{E})[;/L]\ DOP'\ (F2\ \mathscr{F})[;/L]$$

where *F*1 and *F*2 are simple functions of arrays and MOP' and DOP' are similar to MOP and DOP . An example of a function which is not dragged-along by the machine is grade-up which is essentially a sort of its operand. Grade-up obviously does not fit into the above scheme since *F*1 also becomes a sorting function which is not simple as required.

B. Beating and Array Representation

Beating is the machine equivalent of calculating standard forms of select expressions. If the effort to do beating followed by an evaluation of a standard form is less than that to evaluate an expression directly, then the process is worthwhile. We will see in the following chapters that this is in fact the case.

In order to apply beating we must specify a representation of the standard form. One possibility is to maintain the *A*,*F*, and *G* values for each array in an expression to allow calculation of the standard form

A⍉FΔG⍙M

as defined in Chapter II. However, these arrays contain redundant information and it is desirable to find a more compact representation.

If we choose to represent arrays in row-major order we can utilize the representation of the storage access function as the representation of standard forms. In this way, beating will consist of applying the transformations of Chapter II to the mapping functions for arrays.

In the following discussion we can assume without loss of generality that the index origin is zero. Situations where it is different reduce to the zero case by subtracting IORG from all subscripts. Let *A* be a rank-*N* array. Then, assuming that each element in *A* is to occupy one word in memory, the element A[;/L] will be located at

VBASE+(ρA)⊥L (*)

where *VBASE* is the address of *A*[0;0;...;0]. Thus, subscripts of arrays stored in row-major order are representations of numbers in a mixed-radix number system (Knuth [1968] p. 297). This representation is especially suitable for arrays in APL because APL arrays are rectangular, dense, and homogeneous. Further, this representation does not favor any array coordinate over another which is essential in APL.

We can generalize the access function slightly by writing it in the form:

VBASE+*ABASE*++/*DEL*×*L* (**)

where *ABASE* is an additive constant, in this case zero, and *DEL* is the weighting vector used to calculate the base value in (*) above. *DEL* is computed by

DEL[*N*]←1

DEL[*I*]←*DEL*[*I*+1]×(⍴*A*)[*I*+1] for each *I*∈⍳*N*-1.

Example: Let *M* be a matrix with dimension 2,3. Then *DEL*↔3,1 and we set *ABASE*↔0. The layout of *M* in memory is

VBASE ↓	+1	+2	+3	+4	+5
M[0;0]	*M*[0;1]	*M*[0;2]	*M*[1;0]	*M*[1;1]	*M*[1;2]

Given this formulation of the storage access function, it is only necessary to transform *ABASE* and *DEL* in order to obtain the effect of evaluating selection operations on an array.

Example: If *M* is the matrix in the previous example, then the mapping function for (2,1)⍉*M* has the same *VBASE*. For the transpose we use *ABASE*'↔0 and *DEL*'↔1,3. Note that the change in *DEL* corresponds to permuting it by 2,1. This new function uses the same values that were stored for *M*, but accesses them as if they were the transpose (2,1)⍉*M*. To verify this, note that the address for ((2,1)⍉*M*)[*I*;*J*] is

$$
\begin{aligned}
VBASE+ABASE'++/DEL'\times I,J &\leftrightarrow VBASE+ABASE'+(1\times I)+(3\times J)\\
&\leftrightarrow VBASE+ABASE+(3\times J)+(1\times I)\\
&\leftrightarrow VBASE+ABASE++/DEL\times J,I
\end{aligned}
$$

which is the location of $M[J;I] \leftrightarrow ((2,1)\text{⍉}M)[I;J]$.

This can be done for any selection operator by using transformations analogous to those in Chapter II. Appendix A shows the beating transformations on access functions for arrays. In the machine, beating is also applied to expressions containing reductions, scalar operators, and inner and outer products, based on the results in Chapter II.

C. Summary

At this point we have outlined the framework of a machine for APL. It is pleasing to know that it will work since it is justified by theoretical results developed earlier. The remainder of this dissertation discusses the structural details of a machine based on the beating and drag-along processes and gives an evaluation of its effectiveness. Let us outline some goals that such a design should satisfy:

1. The machine language should be close to APL. That is, it should contain all primitives in the language and in a similar form. While it is well-known how to design a machine to accept APL directly there is no particular advantage to doing so. We are primarily concerned with processing the semantics of the language, not its syntax. Thus there is no loss of generality in letting the machine language be a Polish string version of APL. This has the further advantage of freeing the machine from the particular external syntax of APL.

2. The machine should be general and flexible. In particular, it should not be so deeply committed to evaluating APL as to be useless for other purposes.

3. The machine should do as much as possible automatically. This includes storage management, control, and simplification of expressions. The programmer should not have to be aware of the structure and internal functioning of the machine at a level much beyond that specified in an APL program.

4. The machine should do simple things simply and complex tasks in proportion to their complexity. In other words, the work required for the machine to execute a program or expression should be related in some straightforward way to the program's complexity.

5. The machine should be efficient. This is perhaps the most important focus of this work. Of course, the question of efficiency is related to the current technology; at present, a major bottleneck in evaluating array-valued expressions is use of memory. Thus we concentrate on reducing memory accessing and temporary storage space in the evaluation of APL programs.

6. The machine design should be elegant, clean, and perspicuous.

APPENDIX A

TRANSFORMATIONS ON STORAGE ACCESS FUNCTIONS INDUCED BY SELECTION OPERATORS

1. The storage access function for an array M contains the following information:

RANK	↔ ρρ*M*
RVEC	↔ ρ*M*
VBASE	location of first element of ,*M*
ABASE	constant term of access polynomial
DEL	vector of coefficients of access polynomial

Then, the element *M*[;/*L*] is located at

```
VBASE+ABASE++/DEL×L
```

2. This section lists the transformations on storage access functions which are used to effect beating of selection operators. These transformations are given as program segments written in index origin zero. It is assumed that the parameters to the various selection operators are conformable and in the proper domain.

a. *Q↑M*

```
ABASE ← ABASE+DEL+.×(Q<0)×RVEC-|Q
RVEC ← |Q
```

b. *Q↓M*

```
ABASE ← ABASE+DEL+.×(Q>0)×|Q
RVEC ← RVEC-|Q
```

c. *⌽[J]M*

```
ABASE ← ABASE+DEL[J]×(RVEC[J]-1)
DEL[J] ← -DEL[J]
```

d. A⍉M

```
R ← RVEC
D ← DEL
RANK ← 1+(⌈/A)
I ← 0
DEL ← RANK↑DEL
RVEC ← RANK↑RVEC
RANK   REPEAT
   BEGIN
      RVEC[I] ← ⌊/(I=A)/R
      DEL[I] ← +/(I=A)/D
      I ← I+1
END
```

e. M[[J]SCALAR]

```
ABASE ← ABASE+DEL[J]×SCALAR
DEL ← (J≠⍳RANK)/DEL
RVEC ← (J≠⍳RANK)/RVEC
RANK ← RANK-1
```

f. M[[K]J LEN,ORG,S]

```
ABASE ← ABASE+DEL[K]×ORG+(LEN-1)
RVEC[K] ← LEN
IF S=1 THEN DEL[K] ← -DEL[K]
```

CHAPTER IV

THE MACHINE

This chapter contains a functional description of a machine designed to process the semantic content of APL programs.

In general, the description will be given in English, although algorithmic descriptions will be used as necessary to provide clarifications. The section will be written in the style of a programming manual, with the addition of explanations and rationales as required.

The APL machine (APLM) is conceptually composed of two separate machines, each with its own language, sharing the same registers and data structures. The D-machine (DM) accepts APL-like machine code and does all the necessary analysis on expressions. The DM produces code for the E-machine (EM), and in the process does some simplification of incoming expressions using drag-along and beating. The E-machine does all the actual computations of values in the system. By using a stacking location counter based on the organization of machine code into segments, the overall control scheme for the machine is quite simple.

The current chapter consists of five sections which present the APLM in a logical sequence. Section A discusses the data structures and other manipulable objects in the machine, and explains how they are managed in the machine's memory. Section B continues by explaining the stacks and other registers in the machine, followed by a discussion of the overall machine control, in Section C. Finally, the details of the D-machine and the E-machine are set forth in Sections D and E, respectively. Examples are used liberally throughout, to clarify operational details of the APL machine.

A. Data Structures and Other Objects

The manipulable objects in the machine fall into three main classes: data values, descriptors and program segments. This section will describe these three kinds of objects and how they are represented in the machine.

Scalars are the simplest kind of data. In APL, a scalar is an array of rank-0. In practice, a scalar is a different kind of object than an array, and is so treated in the machine. Although arrays are stored in the memory, M, of the machine, scalars are not. They appear only in the machine registers, in particular the value stack, and as immediate operands in a code string. In a real machine, scalars would have an attribute of type, determining the kind of representation to use for encoding and decoding them. In this work, we will assume that this is handled automatically, and that all scalar data are the size of a single machine word.

The most important data structure in the APLM is the array. The representation of an array is divided into two parts. The first is the value array which is a row-major order linearization of the elements of the array. The second part is a descriptor array (DA) for an array, which contains the rank, dimension, and storage mapping function for the array. This separation makes it possible to have multiple DA's, not necessarily identical, referring to the same value array, which makes beating possible. In this chapter, descriptor arrays will be shown in the form:

```
@ARR      RC=2           LEN=05
 +01      VB=VARR        AB=000
 +02         RANK=2
 +03      R(1)=003       D(1)=02
 +04      R(2)=002       D(2)=01
```

@ARR is the address in memory of the first word of the descriptor array for the array named ARR, which is shown above. The first word contains a reference count (RC) and a length (LEN) field, as explained in the discussion on memory in the APLM. The rank of the array is recorded in the third word of the DA; words after that contain the elements of the dimension vector, labeled R(I). Thus in this case, ρARR is 3, 2. The second word in the DA encodes the base address of the value part of the array (labelled VB for VBASE) and the constant term in the storage mapping function (here labelled AB for ABASE). Finally, the DA contains the coefficients of the storage mapping polynomial, DEL (labelled D(I) here). Recall that for an array ARR, the element ARR[;/L] is located at

$$\mathrm{VBASE} + \mathrm{ABASE} + +/\mathrm{DEL} \times (\mathrm{L} - \mathrm{IORG});$$

This formula is the storage mapping function for any array.

In addition to array descriptors, the machine contains descriptors for J-vectors. Recall from Chapter II that a J-vector is a vector of consecutive integers which can be specified by a length, an origin, and a direction bit. We assume that these three quantities can be encoded into a descriptor by the function JCODE(length, origin, direction) and that there are appropriate decoding functions. (See Appendix A.)

Finally, programs in the machine are represented internally as program segments. A program segment is any sequence of machine commands and operands, and is referenced by a segment descriptor. Segment descriptors contain an encoding of the beginning address of a segment (relative to the beginning of the function they are a part of) and the length of the segment. There is also a bit which indicates the execution mode for the segment (see Section C).

Each defined function (program) is a segment, and logical subparts of the function may also be represented as segments. As will be seen later, it is easy to activate and de-activate segments in the APL machine. Briefly, the advantages of organizing programs in segments is that these are the logical units of a program, while other organizations, such as paging, do not allow this kind of natural correspondence of form and function (pardon the pun!). An important property of APLM instructions is that they contain no absolute addresses except for references to NT, which remain constant in any compilation. All internal references to other parts of a program are relative. Thus, all programs are relocatable.

Each function has a corresponding function descriptor, which is similar to a DA. A function descriptor contains the following information:

FVBASE	location in M of beginning of function segment
FLEN	length of function segment
FIORG	index origin for this function
FISR	logical variable ←1 if function has a result
FPARS	number of parameters
FLCL	total number of local names

In addition, the rest of the function descriptor contains a list of all local names in the function, in the order: result (if any), parameters (if any), local variables (if any). The function descriptor for a function is used in calling and returning from functions, as will be discussed in Section D.

Main memory in the machine is a linear array of words named M. The only objects which reside in M are arrays, DA's, and program segments. All other objects are stored in the machine's registers. In addition to M, there is an array NT, the Nametable, which is an abbreviated symbol table. Every identifier in the active workspace has an entry in NT, which contains descriptive information and either an actual value or a pointer to where it can be found in M. Scalars and J-vector descriptors are stored directly in NT. Thus, all references to variables and functions in the machine go through the NT. This organization allows for dynamic allocation and relocation of space in M, without having to alter any program references. The operation of NT is described more fully in the next section under machine registers. Constant array values within a function are stored as part of the program segment; they are addressed relative to the beginning of the function, and so, too, remain relocatable.

Within M, two different allocation mechanisms are used, one for functions and array values, and one for descriptor arrays. The reasons for this are that, because of drag-along and beating, DA's are expected to have a shorter lifetime than functions or array values. Further, in a given function, locally at least, it is likely that DA's will be of similar sizes. Thus, it is feasible to keep an available space list for DA's, with the hope that erased spaces can be reused intact. We would therefore expect more efficient use of M by DA's than by array values.

The free memory space (M) is arranged as follows: functions and array values are allocated from the lowest address (BOTM) towards the top of M and DA's are allocated from the top (TOPM) down. The space in the middle is the POOL, with boundaries BOTP and TOPP. Each entry in M has a header word containing an encoding of a reference count (see Collins [1965]), the length of the entry, and a filler count. The latter field is used when space slightly larger than necessary is allocated. Each time a reference to an entry is added or deleted, the reference count field is adjusted. When a reference count goes to zero, meaning that there are no uses of the entry anywhere in the system, the entry is made available in ono of two ways. If it is adjacent to the POOL, it is merged with POOL. Otherwise, it is added to the appropriate availability list, of which there are two, one for DA's and one for functions and array values.

The availability lists are doubly linked, and each entry contains a header similar to those for active entries. When space is needed, the appropriate availability list is searched using the first-fit method (Knuth [1968] 436, ff). If a fit is found, the space is allocated and the availability list adjusted. Otherwise, space is taken from the POOL. If a request for M-space is made which cannot be honored because there is not enough contiguous space available, a garbage collection is made. The two halves of M are garbage-collected separately. In collecting array space, all the DA's are scanned and a linked list is set up which ties together all DA's pointing to the same entry. Then arrays are compacted towards BOTM, with the links used to adjust the VBASE fields in the referent DA's. If enough space is still not available, the DA's are also compacted, using a similar algorithm. Some coalescing of available space is also done by the allocation algorithm, GETSPACE. Figure 1 illustrates how M is structured.

B. Machine Registers

This section describes the registers and register-like structures in the APL machine. The present description covers only the logical functions performed by these registers and does not make any demands on how they are actually to be implemented. Although most of the registers are not directly accessible to the programmer, thorough knowledge of their use is important to understanding the functioning of the machine.

There are several registers related to memory accessing and allocation. The most important of these is the Nametable, NT. NT is an associatively addressed stack, each entry of which contains a name field, a tag, and a value. The name field of an entry contains an index for the identifier associated with the entry. Permissible tags in NT are ST, for scalar quantities, JT, for encoded J-vectors, UT, for undefined identifiers, DT, for arrays, and FT for functions. ST and JT entries contain the actual value in their value field, while DT and FT entries have descriptor addresses in their value fields.

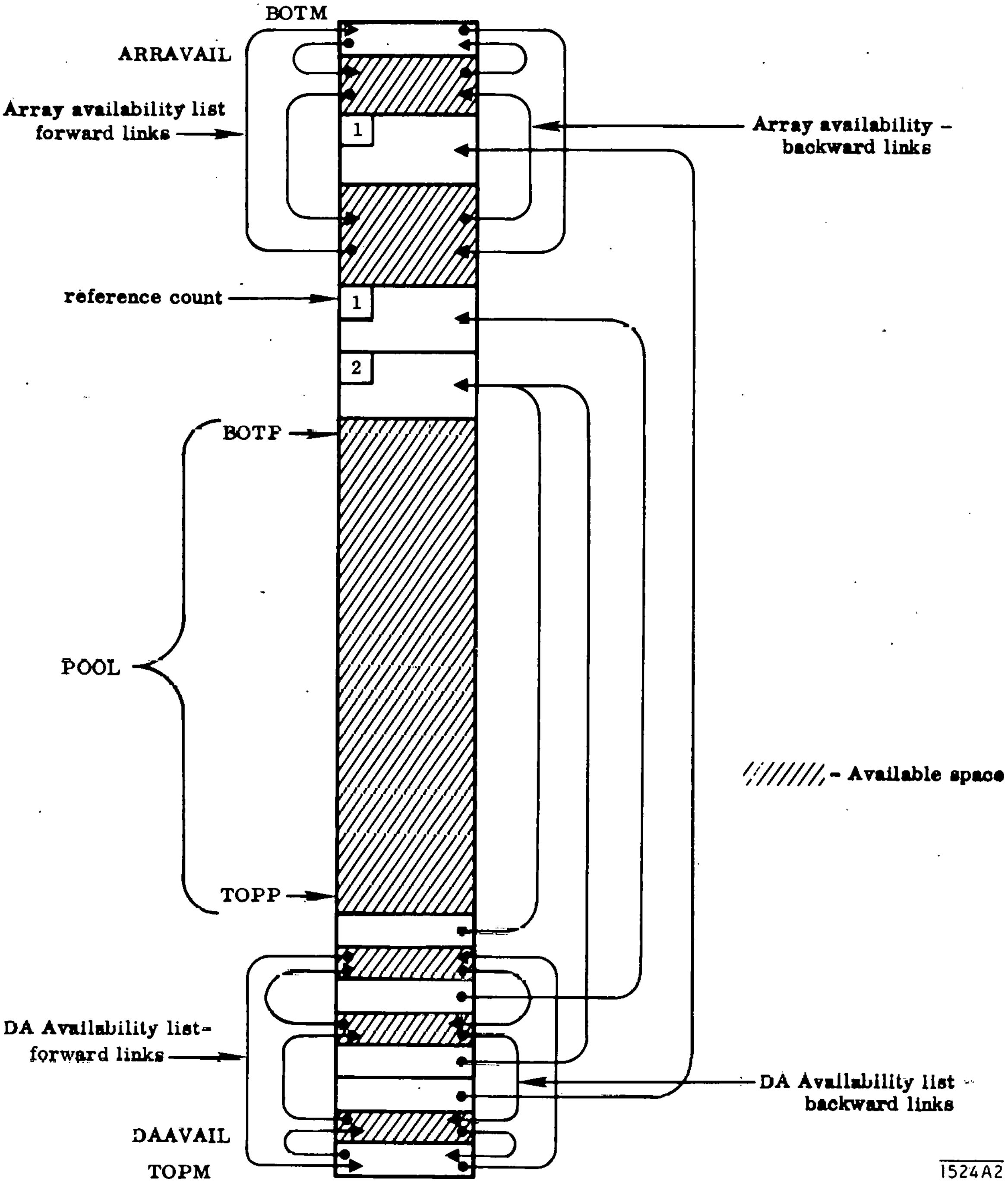

BOTM
ARRAVAIL
Array availability list forward links
Array availability - backward links
1
reference count
1
2
BOTP
POOL
TOPP
▨ - Available space
DA Availability list- forward links
DA Availability list backward links
DAAVAIL
TOPM
1524A2

FIGURE 1--Structure of M.

When a function is called, an entry is pushed to NT for each of the function's local variables and parameters, as listed in the function descriptor. Similarly, when a function is de-activated, the reverse process occurs. Each time a variable is accessed, NT is searched associatively from the top (latest entry). If a hit is not found, then the desired variable must be global, and it is entered into NT. This mode of maintaining the NT makes identifier behavior correspond to APL's "dynamic block structure" and facilitates recursive function calls.

The most important registers in the APL machine are four stacks. The use of stacks permits elimination of addresses from most instructions and simplifies the evaluation of recursive and nested programs.

1. Value Stack (VS)

VS is the main stack in the machine and is used in the evaluation of expressions and in function calls. Each VS entry consists of a tag and a value part, as in NT entries. In addition to scalars and function or DA pointers, VS can contain segment descriptors, partially-evaluated addresses, function marks, and names.

2. Location Counter Stack (LS)

Recall that machine code is organized into segments, characterized by a starting address and a length. Each LS entry contains the starting address of a segment (ORG), its length (LEN), a relative count, pointing to the next instruction to be executed (REL), and control information. Each time a segment is activated, its beginning address and length are pushed to LS, and the REL field is set to zero. The address of the next instruction is then determined from the REL and ORG fields on the top of LS. After each instruction fetch, the REL field at the top of LS is incremented. When this value is equal to the length of the segment, the segment is terminated by popping the top of LS, thereby reactivating the next entry. The control information in LS is used to coordinate it with the other stacks in the machine.

3. Iteration Control Stack (IS)

Array-valued APL expressions implicitly specify an index set for the expressions. In this machine, IS is used to control (nested) iterations over this index set in the element-by-element evaluation of array-valued expressions. The operation of IS is coupled with LS as follows: when a set of iterations is begun, the limits of the iteration are pushed into the iteration stack, and a segment is activated containing the range of the iterations. Then, for each instruction in the code segment, the necessary index values are taken from IS. When the segment is completed, the entries in IS are stepped and if the required iterations are not exhausted, the segment is re-initialized and repeated with the new IS values. Eventually, the iterations are completed and the segment in the range also is completed, in which case IS and LS are both popped, returning the machine to the place it was to resume after the iterated code was completed. (See Section D.)

The IS behaves essentially like a nest of FORTRAN DO's. Each entry contains a counter (CTR) (to origin zero), the maximum value of the counter (MAX), direction bit (i.e., count up or down) (DIR) and control information. Although the IS is partially accessible to the machine code, it is for the most part maintained automatically. Like LS, IS could probably be incorporated into the value stack, since these three stacks generally work in parallel. However, by separating these stacks by their functions, the machine design becomes cleaner and more perspicuous.

4. Instruction Buffer (QS)

Unlike LS and IS, the instruction buffer QS is logically separate from the value stack. QS is not strictly a stack, since it is possible to access and alter information at places other than its top. In the D-machine, instructions are fetched from M, some of which are executed immediately, and others of which are either evaluated by beating or are deferred in QS by drag-along. In entering instructions in QS, the DM may change other related QS entries. When the E-machine is activated, instructions are fetched from QS and executed directly, generally in conjunction with VS and IS. QS contains operation and value fields, similar to VS, a LINK field used to reference other deferred instructions, and an AUX field, which is a logical vector acting as an access mask for array entries (see Section E).

A final four registers in the machine are mentioned primarily for completeness. These are:

IORG	Index origin of current active function
FBASE	Base address in M of current active function
FREG	VS index of function mark for current active function
ISMK	IS index of topmost IS entry containing 1 in its MARK field.

The use of these registers is shown in the examples in following sections.

C. Machine Control

The purpose of the APL machine is to transform a set of data (the input) into a second set (the output) according to encoded transformation rules (the program) which are interpreted according to a predetermined scheme (the machine). This entire process is called the evaluation of the program and input.

In the APL machine, programs are evaluated in two separate but related sub-machines. The D-machine takes its instructions from main memory, M, in the form of Polish APL code, and does all the necessary domain testing and storage allocation for the various operands. In addition, the DM does simplification of incoming expressions by drag-along and beating. The output of the D-machine is values in VS and transformed code in the QS, in the form of instruction segments for the E-machine. At critical points, determined either by the programmer and the DM, control is passed to the E-machine, which executes the simplified instructions in QS, producing values in VS and M. When done, the EM passes control back to the DM, which resumes where it left off.

The division of labor between the two submachines is logically similar to that between a compiler and its target machine. The DM plays the role of the algebraically simplifying compiler, whose source language is essentially APL, and whose target language is E-machine code. The E-machine as the target of the DM's transformations is a conceptually simple computer which does nothing but compute values. Given this scheme, a question which naturally arises is, Why bother with the D-machine at all? Why not use a separate compiler in software and let it produce code for a machine similar to our E-machine? Unfortunately, this is impossible, since the behavior of the D-machine is dependent not only on the source code (program), but is also dynamically dependent on the data. For instance, consider a simple APL expression such as A + B. We would like the source code for this expression to be something conceptually like

```
LOAD  B    (i.e., "load" B to the value stack)
LOAD  A
ADD        (i.e., add the values on top of the value stack and leave the
            result there.)
```

The problem here is that we would like the machine to do different things depending on the data. In particular, if both A and B are scalars at the time the above code is executed, it would be desirable to have the LOAD instructions push the actual scalar values to the stack, and to have the ADD do the actual addition. But if A and B are conformable arrays, the desired action is to defer the entire operation (both LOADs and the ADD) in the instruction buffer, to be performed later by the E-machine.

No compiler would be able to make these decisions a priori unless it knew what data was to be used in running the program, or unless variables were sufficiently restricted by declarations. Further, much of the work done by the D-machine is domain testing, including rank and dimension checking, on dynamically-specified variables. Since this process is data-dependent, it must be performed dynamically.

Both the D-machine and the E-machine share all the registers and the memory of the entire APL machine. Further, both are controlled by a central cycle routine, shown in Fig. 2. The key to the overall control of the APLM is the location counter stack, LS, which contains active segments for both the DM and the EM. In Fig. 2 we see that a major machine cycle takes the form:

a. Check to see if the current active segment has been completed. If not, proceed to step b, otherwise see if this segment is under control of the iteration stack. If it is, then step the iteration stack; in case IS does not overflow, then reset the REL field to the beginning of the segment and repeat this step. If the segment is not under control of IS or if it is and the iteration stack overflowed, then de-activate the segment and repeat this step.

b. Calculate the effective address of the current instruction and update the location counter stack.

c. Select the appropriate machine, determined by the D/E bit in the current active segment. If the DM is selected, then defer any arrays referenced on the top of the value stack to the instruction buffer; also, fetch the instruction and (if necessary) the second word of the instruction from memory. Finally, decode and interpret the instruction and return to step a.

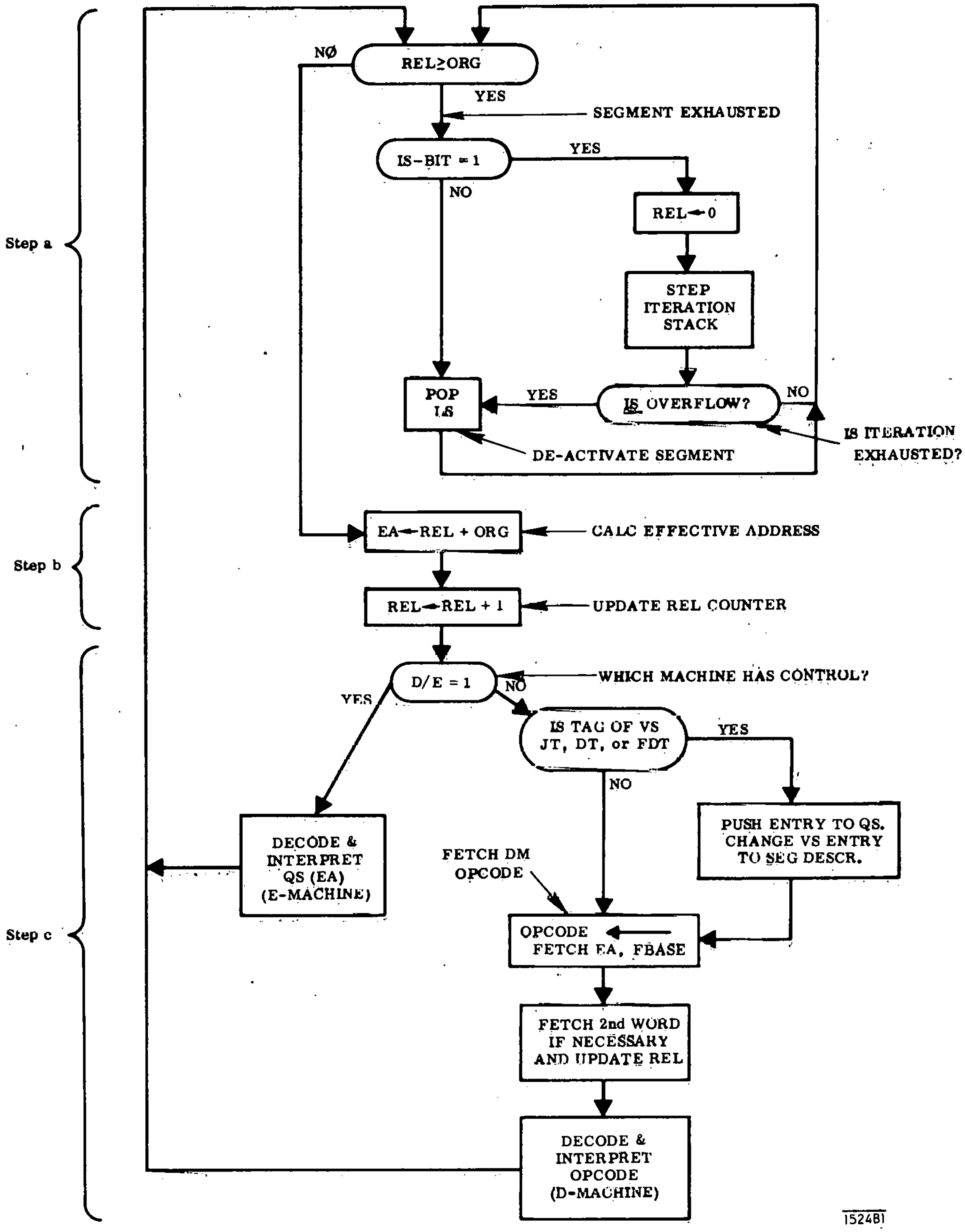


FIGURE 2--Maincycle routine.

D. The D-Machine

The D-machine evaluates programs written in "machine language" by generating instructions in QS to be executed later by the E-machine. As discussed in Chapter III, the use of a Polish string for the machine language rather than "raw" APL frees the APLM from the particular concrete syntax of APL without sacrificing any of the semantic content.

Most of the instructions in the APLM correspond directly to the APL primitives; those which do not are the control instructions, which comprise a more powerful set in the machine than are provided in the source language. All operands in DM instructions are either relative addresses within the program segment or are NT references or are immediate values. As a result, all programs in the machine are relocatable. Since only constant data is contained in function segments, programs are likewise re-entrant.

The D-machine instruction set is listed in Tables 1-1, 1-2, and 1-3. The instructions are divided into three classes: storage management instructions, control instructions, and operator instructions. It is clear from Table 1 that no systems functions are included in the D-machine's repertoire. In a real implementation of an APL machine, these instructions would have to be provided, although for the current work, they are irrelevant. The remainder of this section discusses the instructions of the D-machine, with examples to clarify the details.

0. A Guide to the Examples

The examples used in this chapter include program listings, register dumps, and memory dumps. In showing program excerpts, we generally also show the APL source expression, and give values, or at least attributes, for the operands. Programs are shown in assembly language format, except that absolute addresses are given. Although nothing has been said of the manner in which D-machine instructions

TABLE 1-1

Storage Management and Control Instructions

Opcode	Operand	Description
A. Storage Management Instructions		
LDS	scalar	Load scalar
LDSEG	seg-descr	Load segment descriptor
LDJ	jcode 1, o, s	Load J-vector
LDIS	K	Load iteration stack counter, K from top of IS
LDCON	K	Load constant array, starting at FBASE +K
LDN	N	Load name N
LDNF	N	Load name N and fetch value
ASGN		Assign (and discard value)
ASGNV		Assign and leave valuc
B. Control Instructions		
JMP	K	Jump by K (signed) in current segment
JMP0	K	Jump by K in current segment only if top of VS is 0 Pop VS in either case
JMP1	K	Same as JMP0 except test for 1
LEAVE		De-activate this segment (i.e., pop LS and also IS if necessary.)
RETURN		Return from current function
ITM		Iterate and mark
DO		Call E-machine to work on top of VS
DOI		Same as DO except that temporary space is allocated for the result, if any, and the result is left on top of VS

TABLE 1-2

Scalar Arithmetic Operators

Operator	APL	Definition
A. Dyadic		
ADD	+	Add
SUB	-	Subtract
MUL	×	Multiply
DIV	÷	Divide
MOD	|	Modulus
MIN	⌊	Minimum
MAX	⌈	Maximum
PWR	⋆	Power
LOG	⍟	Logarithm
CIR	○	Circular functions
DEAL	?	Random deal
COMB	!	Binomial coefficient or beta function
AND	∧	Logical and
OR	∨	Logical or
NAND	⍲	Logical nand
NOR	⍱	Logical nor
LT	<	Less than
LE	≤	Less than or equal
EQ	=	Equal
GE	≥	Greater than or equal
GT	>	Greater than
NE	≠	Not equal
B. Monadic		
PLUS	+	Plus
MINUS	-	Minus
SGN	×	Signum
RECIP	÷	Reciprocal
ABS	|	Absolute value
FLOOR	⌊	Floor
CEIL	⌈	Ceiling
EXP	⋆	Exponential (base e)
LOGE	⍟	Logarithm (base e)
PI	○	Pi times
RAND	?	Random number
FAC	!	Factorial or gamma function
NOT	~	Logical not

TABLE 1-3

Remaining Operators in D-Machine

Operator	APL	Definition
A. Selection		
TAKE	↑	Take
DROP	↓	Drop
REV K	ϕ[*K*]	Reverse along $K^{\underline{th}}$ coordinate
TRANS	⍉	Generalized transpose
INX K	[[*K*]	Index on $K^{\underline{th}}$ coordinate
B. Evaluated Immediately		
BASE	⊥	Base value (Decode)
REP	⊤	Representation (Encode)
GDU	⍋	Grade up
GDD	⍒	Grade down
CAT K	,	Catenate (top K on VS)
RAV	,	Ravel
URHO	ρ	Dimension
DRHO	ρ	Restructure
UIOTA	ι	Interval
C. Deferrable		
ROT K	ψ[*K*]	Rotate on $K^{\underline{th}}$ coordinate
EPS	ϵ	Membership
DIOTA	ι	Rank
CMPRS K	/[*K*]	Compress on $K^{\underline{th}}$ coordinate
EXPND K	\[*K*]	Expand on $K^{\underline{th}}$ coordinate
SUBS K	[	Subscript with K expressions in VS
D. Compound		
RED K OP	*OP*/[*K*]	Reduce along $K^{\underline{th}}$ coordinate by OP
GDF OP	---	General dyadic form with OP

are encoded, we have chosen, for purposes of illustration, to show them as one or two word quantities, depending on whether or not they have operands. All operand addresses are shown symbolically and comments are used to explain the program structure. In the register dumps, most of the material is self-explanatory. Field headings are summarized in Appendix A. The top of each stack is indicated by an arrow. Descriptor array addresses, which are pointers to the memory, are in the form @A, for variable A, and value addresses in M are of the form VA. Again, in the real machine, these would in fact be numerical addresses, but the symbolic form is much clearer for examples. Fields in DA's are labelled mnemonically. Segment descriptors in VS or QS are shown in the form SCODE(SEG.X, m), where m is 0 or 1 depending on whether the segment is a DM or an EM segment, and X is the segment symbolic name (arbitrary). EM segments are delimited by "brackets" along the right side of the QS display, in the format XY, meaning that segment X starts here and segment Y ends here. The LINK field of QS contains relative pointers and is interpreted according to the opcode. The contents of the AUX field is to be interpreted as a logical vector, although in fact it may be encoded differently in an actual APLM.

1. Storage Management Instructions

This class includes all instructions concerned primarily with the storing and fetching of data. Each of the load instructions pushes a value to the value stack. Of these, four have immediate operands; LDS, LDSEG, LDJ, and LDN push their operands to VS with tags ST, SGT, JT, and NPT respectively. LDIS K loads as a scalar the current value of the CNT field of the iteration stack element K entries from the top of IS. LDNF N refers to variable N in the nametable, and enters the current value of the variable (from NT) into VS. In the case of NT entries with tag DT (i.e., arrays), the reference count of the DA is increased by 1 when it is entered into VS, and the VS tag is set to FDT. The LDCON K instruction is used to access a constant array stored in a function segment. Its operand K is a pointer relative to the function origin pointing to the beginning of the DA for the constant value. This DA is copied to the DA area of M, its VBASE is set to the beginning of the function (FBASE), and its ABASE is set to K. The DA pointer is pushed to VS with tag FDT.

Although all the load instructions just described push a value to VS, such values do not always remain there. At the beginning of each D-machine cycle, the top of VS is examined for tags FDT, DT, and JT (see Fig. 2). If one of these is present, then the entry is deferred in QS, because it is array-valued. This is done by pushing an E-machine instruction to QS of the form

OP @ARR 0 MASK.

OP is IFA, IA, or IJ, depending on whether the VS tag was FDT, DT, or JT; @ARR is the DA pointer that was in the VS value field, and MASK is an access mask. The access mask in this case is a logical vector whose last K bits are 1 when ARR is a rank-K array. It will be used by the DM in beating and by the EM in accessing this array. The LINK field in E-machine instructions of this type is unused, and thus is shown as 0 above. The VS entry is then replaced by a segment descriptor with tag SGT pointing to the one-word QS segment containing the deferred operand. In general, this entire process is invisible in the examples below, and the load instructions which generate array values can be thought of as doing the deferral themselves.

Although ASGN and ASGNV are operators, they are included as storage management instructions because they have the side-effect of causing values to be stored. These instructions expect the top of VS to contain a destination, either as a name (tag NPT) or as a QS descriptor pointing to a segment containing only

TABLE 2

Interpretation of ASGN and ASGNV in the D-Machine

	Top of VS	(Top-1) of VS	Action
a.	tag = NPT or tag = SGT and deferred expression has one element	tag = ST	Do immediate assignment. That is, store the scalar value in NT or in M, as appropriate.
b.	tag = NPT	tag = SGT and deferred segment is a J-vector	Do immediate assignment.
c.	tag = NPT	tag = SGT and deferred segment is a single DA with reference count of 1 and value also has reference count of 1	Do immediate assignment.
d.	tag = NPT	tag = SGT and deferred segment is any arbitrary array expression	Allocate space for a DA and value of the size necessary to store the result. Defer the assignment in QS, as for scalar arithmetic operators.
e.	tag = SGT and deferred segment consists of a QS entry with opcode IA	tag = SGT and deferred segment is any arbitrary array expression	Check ranks and dimensions for conformability. If the lhs variable is a J-vector, it must first be explicitly evaluated. If the rhs expression contains instances of the lhs variable with different permutations, then the rhs expression is evaluated to temporary space. Finally, the assignment is deferred as above.

an IA instruction; the second entry in VS is the right-hand side of the assignment. There are several possible actions taken by the DM in interpreting assignments, depending on the VS contents. These cases are explained in Table 2. We have assumed that "evil" side effects do not appear in the code; their treatment is straightforward, but uninteresting. Also, it should be noted that although the strategies outlined in Table 2 could be modified to alter the machine's performance, the case analysis remains the same.

The final storage management instructions are INPUT and OUTPUT, which are left further unspecified. These could be conceived of as read-only and write-only (serial) strings, which are used as primitives for writing functions such as ⎕ and ⍞ .

2. Control Instructions

The control instructions of the APLM are all concerned with directing the flow of control among statements at the source-language level, and are all evaluated by the D-machine.

The three jump instructions, JMP, JMP0, and JMP1 are used to alter the flow of control among statements in a function. Since no jumps are allowed outside of a function, there is little difficulty in specifying this operation. All that is necessary is to change the value of the relative pointer in the current segment on LS. CYCLE is a special case of JMP, which sets the relative pointer to 0, causing the current (D-mode) segment to be repeated. LEAVE pops LS and also IS, if the segment is involved in an iteration. RETURN performs similarly in returning from a call on a function. In addition, it automatically erases the locals for the current function from NT.

The interpretation of the DO instruction depends on the top value on VS. If the top of VS is a scalar then the DO acts as a no-op. If the tag is SGT, then the segment described on VS is activated by pushing the segment descriptor to LS, with VS being popped. In case the tag is NPT, the corresponding NT tag is examined, and if the tag is FT, then the named function is activated, as described in the next paragraph; all other cases are no-ops. The DOI instruction is similar to DO except that if the top is VS and has tag NPT, the value referenced is copied to newspace, while if the tag is SGT, temporary space is allocated for the result and the segment is evaluated. Thus, after executing a DOI, the top of VS contains an entry with tag ST, JT, or FDT.

When a DO instruction encounters a function name on top of VS, the following actions take place:

1. The function descriptor, referenced by the NT entry for the function, is fetched. It is expected that all parameters to the function have been evaluated and placed on top of VS, so that the topmost value is the leftmost parameter. The parameter count, FPAR, in the function descriptor is fetched, and the top of VS checked to see that there are that many values already there. If not, an error is signaled. Otherwise, the machine goes through the list of local variables in the function descriptor, making an entry in NT for each one. Each new tag in NT is set to UT, for undefined, unless it corresponds to a parameter. Parameter values are placed in NT and popped from the value stack in order.

2. A function mark entry is pushed to VS, with tag FMT containing an encoding of the current values of FREG, IORG, and the name of the function being activated.

3. IORG is set to the value in the function descriptor, and FREG is set to the VS index of the function mark.

4. An entry is pushed into LS for the segment described by FVBASE and FLEN in the function descriptor. FBASE is initialized to FVBASE, and the process is completed.

The segment just activated contains all the code for the function. When a RETURN is executed within this function, the following occurs:

1. LS is popped, thereby de-activating the function.

2. The function name, encoded in the function mark onVS, is used to access the function descriptor and then popped. If there is a result, the value is pushed to VS, and its NT entry erased. All other NT entries for locals in the function, together with their values, are also erased.

3. FREG and IORG are restored from the values in the function mark on VS. The function mark is deleted and the result, if any, is moved into its place.

4. Finally, FBASE is set to point to the current active function (if any) by accessing its function descriptor through its name in the newly-exposed function mark.

3. Operator Instructions

The operator instructions correspond to the primitive operators in APL. They can be considered in four groupings, and are so discussed in the rest of this section. Part a discusses the scalar arithmetic operators (Table 1-2); part b contains a description of the selection operators which are evaluated by beating (Table 1-3A); part c describes those operators which are generally executed immediately (Table 1-3B); and part d covers remaining deferrable operators as well as the compound operators (Table 1-3C, D).

a. Scalar arithmetic operators

If the top of VS contains two scalar values (or one if the operator is monadic) then the operation is done immediately, leaving a result in VS and popping the operand(s). This process is illustrated in Example 1. In fact, the operation is pushed to QS and the E-machine is activated to perform the actual evaluation, but this micro-process is invisible to the user.

The other possible cases occur when the top two elements of VS are segment descriptors for deferred code in QS or when one is a segment descriptor and the other is a scalar. If one of the operands is a scalar, it is entered into QS and its VS entry is replaced by an appropriate segment descriptor, reducing it to the case of two segment descriptors in VS.

The D-machine compares the ranks and dimensions of the two operands for conformability and signals an error if they don't match. Otherwise, the operation is deferred by drag-along in QS and the top of VS adjusted so that it contains a segment descriptor pointing to the entire deferred expression in QS. Because of the stack discipline in the machine, the deferred code for both operands will always be contiguous in QS. The link field of the QS entry for the operator (with opcode OP) is a relative backwards pointer to the earliest deferred operand in the deferred subexpression. The AUX field is the same as the AUX field of the two operands (see Example 2).

b. Selection Operators

The selection operators are evaluated in the D-machine by beating, the process of performing a selection operation on an array-valued expression by changing the storage mapping functions of its constituent array operands. The mathematical analysis of Chapter II legitimizes this approach, and guarantees that the transformations used in beating produce the correct results. Before proceeding, let us define what it means for an array-valued expression to be beatable.

An array-valued expression deferred in QS is <u>beatable</u> if any of the following conditions apply:

(i) It is a single QS entry with opcode IFA or IJ.

(ii) It is a consecutive pair of QS entries of the form

S	scalar	0	0
IRD	ptr	0	R .

```
      EXAMPLE 1 - SCALAR OPERATOR, SCALAR OPERANDS
--------------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 00200

           REL   ORG   LEN  D/E IS  FN  NWT  QP
      LS: +-----+-----+-----+---+---+---+---+-----+
          | 010 | 00C | 10C | 0 | 0 | 1 | 0 | C0 |
      --> |

          EFFECTIVE ADDR = 0210   IN M

           TAG   VALUE                      OP    VALUE                LINK  AUX
      VS:+-----+--------------------+  QS:+-----+--------------------+-----+-------+
         | ..  |       ...          | --> |
         | ST  | 256                |
         | ST  |  32                |
      -->|
```

EXAMPLE 1-1: BEFORE EXECUTING ADD AT M(21C)

```
--------------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 0      IORG = 0      FREG = 00C00      FBASE = C0200

           REL   ORG   LEN  D/E IS  FN  NWT  QP
      LS: +-----+-----+-----+---+---+---+---+-----+
          | 011 | C00 | 100 | 0 | 0 | 1 | 0 | 0C |
          | 000 | C00 | 001 | 1 | 0 | 0 | 0 | 0C |
      --> |

          EFFECTIVE ADDR = 0000   IN QS

           TAG   VALUE                      OP    VALUE                LINK  AUX
      VS:+-----+--------------------+  QS:+-----+--------------------+-----+-------+
         | ..  |       ...          |  00 | OP  | ADD                |     |       |
         | ST  | 256                | --> |
         | ST  |  32                |
      -->|

           THE ADD INSTRUCTION AT M(210) HAS BEEN FETCHED, DECODED,
           AND DEFERRED IN QS.  SINCE BOTH OPERANDS ARE SCALARS,
           THE DEFERRED SEGMENT IS ACTIVATED IMMEDIATELY. (NOTE L3)
```

EXAMPLE 1-2: AFTER DECODING ADD; OPERATION DEFERRED IN QS

```
      EXAMPLE 1 - SCALAR OPERATOR, SCALAR OPERANDS
--------------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 002C0

           REL   ORG   LEN  D/E IS  FN  NWT  QP
      LS: +-----+-----+-----+---+---+---+---+-----+
          | 011 | 000 | 100 | 0 | C | 1 | 0 | 00 |
          | 001 | 000 | C01 | 1 | 0 | 0 | 0 | 00 |
      --> |

          EFFECTIVE ADDR = 0001   IN QS

           TAG   VALUE                      OP    VALUE                LINK  AUX
      VS:+-----+--------------------+  QS:+-----+--------------------+-----+-------+
         | ..  |       ...          |  00 | OP  | ADD                |     |       |
         | ST  | 288                | --> |
      -->|
```

EXAMPLE 1-3: AFTER E-MACHINE EXECUTION OF ADD; QS SEGMENT EXHAUSTED

```
--------------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 0020C

           REL   ORG   LEN  D/E IS  FN  NWT  QP
      LS: +-----+-----+-----+---+---+---+---+-----+
          | 011 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
      --> |

          EFFECTIVE ADDR = 0213   IN M

           TAG   VALUE                      OP    VALUE                LINK  AUX
      VS:+-----+--------------------+  QS:+-----+--------------------+-----+-------+
         | ..  |       ...          | --> |
         | ST  | 288                |
      -->|
```

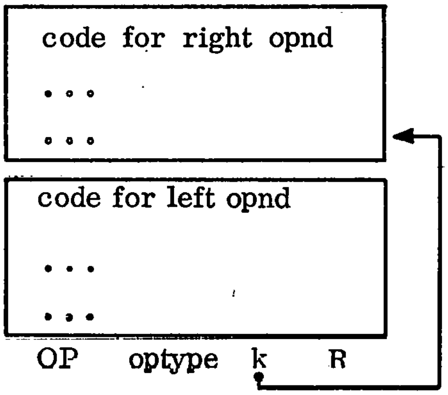
EXAMPLE 1-4: AFTER RETURN TO D-MACHINE. RESULT OF ADD IS ON VS

```
EXAMPLE 2 - SCALAR OPERATOR, ARRAY OPERANDS
-----------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 010 | 000 | 1C0 | 0 | 0 | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0210   IN M

     TAG   VALUE                    OP    VALUE             LINK  AUX
VS:+-----+------------------+  QS:+-----+------------------+----+------+
   | ..  |       ...        |  00 | IFA | @A               |    | 0111 | AA
   | SGT | SCODE(SEG.A,1)   |  01 | IFA | @B               |    | 0111 | BB
   | SGT | SCODE(SEG.B,1)   | --> |
-->|

     ARRAYS WITH DA'S AT 1000 AND 1010 ARE OF RANK 3 (NOTE QS AUX FIELDS).
     NEXT INSTRUCTION IS  ADC  AT M(210)

EXAMPLE 2-1: BEFORE EXECUTING ADD

-----------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 0020C

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 011 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0211   IN M

     TAG   VALUE                    OP    VALUE             LINK  AUX
VS:+-----+------------------+  QS:+-----+------------------+----+------+
   | ..  |       ...        |  00 | IFA | @A               |    | 0111 | C_
   | SGT | SCODE(SEG.C,1)   |  01 | IFA | @B               |    | 0111 |
-->|                           02 | OP  | ADD              | 02 | 0111 | _C
                              --> |


EXAMPLE 2-2: AFTER DEFERRING ADD
```

(iii) It is a QS segment consisting of a scalar monadic operator operating on a beatable sub-segment. That is, it is of form:

```
┌──────────────────────┐
│ code for operand     │
│ ...                  │
│ ...                  │
└──────────────────────┘
 OP optype   1     R
```

(iv) It is a QS segment consisting of a pair of beatable operands combined by a dyadic scalar operator. One of these operands can optionally be a scalar value. The form is:



```
┌──────────────────────┐
│ code for right opnd  │
│ ...                  │
│ ...                  │
└──────────────────────┘
┌──────────────────────┐
│ code for left opnd   │
│ ...                  │
│ ...                  │
└──────────────────────┘
 OP   optype   k    R
```

(v) It is a pair of beatable operands combined by GDF. The form is similar to case (iv) above.

(vi) It is a reduction of a beatable operand, in the form:

```
    BRED  0       k   0
    ┌──────────────────────┐
    │ code for reducee     │  A_
    │ ...                  │
    │ ...                  │
    └──────────────────────┘
    OP reduce-op             _A
    SGV SEG.A
k:
    S  -length
    ITM
```

(vii) In addition to (i) through (vi) above, a single QS entry with opcode IA is beatable, although it does not enter into the recursive definition.

When a selection operation is interpreted by the D-machine, the array-valued operand is first checked for conformability. If the operand is beatable, then it is beaten, according to the transformations shown in Chapter III, Appendix A. In this process, if a DA to be transformed has a reference count of 1, indicating that it is a local temporary result, then the DA can be modified directly. If the reference count is greater than 1, then a copy must be made, and the copy is beaten. If the result of a beating operation is a scalar value, then the segment is turned over to the E-machine, which evaluates it and leaves the scalar result on the top of VS.

When the operand of a selection operation is not beatable, there are two possible strategies to follow: In the case of the TRANS operation, there is no choice: the operand must be evaluated by the E-machine and a temporary value stored, which is then beaten as above. Otherwise, the selection operation can be treated as a special case of subscripting, in which case an appropriate set of E-machine instructions is dragged-along in QS. (See Section d. for an explanation of subscripting.) The choice of strategies is a second-order design decision, and need not be made at this time, since either approach is viable. Example 3 illustrates both beating of selection operators and drag-along of scalar operators.

The DM code shown for the statement is a straightforward translation of the APL statement into Polish. Note that the vector 2,¯2 is a constant and is "compiled" into the function segment. This approach avoids having to keep array-valued constants in the memory with other array quantities; to do so would require having an entry in NT for each such constant, and would complicate the storage management functions. In Examples 3-1 and 3-2, the state of the machine before executing the sample code is shown; the values of the variables M and N are not

EXAMPLE 3: DRAG–ALONG AND BEATING IN THE D–MACHINE

Consider the APL expression

R←(2,1)⍉(⌽[1]M)+(2,¯2)↑N

At the time this is to be evaluated, ρM↔2,2 and ρN↔3,4 . Assume that R has no current value. The machine code for this statement is shown as follows, starting at location 250 in memory.

Addr	Op	Operand	Comments
250	LDNF	N	
252	LDCON	90	Refers to constant 2,¯2 with DA at 290
254	TAKE		
255	LDNF	M	
257	REV	0	(Recall 0-base in all machine code)
259	ADD		
260	LDJ	JCODE(2,1,1)	This is the vector 2,1
262	TRANS		
263	LDN	R	
265	ASGN		Assign (and discard value)
266	. . .		
. . .			
. . .			
290	RC=1	LEN=4	DA header
291	VB=0	AB=94	DA for constant vector 2,¯2. See Section A for description of format.
292	RANK=1		
293	R(1)=2	D(1)=1	
294	RC=1	LEN=3	Header for value array
295	2		Value
296	-2		

given, as they are irrelevant for this example. LS contains a descriptor for a D-machine segment of length 100, which is the main segment of the function F. The effective address is the sum of the REL field of LS and FBASE, the beginning of the value part of function F. VS contains a function mark for F which was placed there when F was called.

In 3-3 and 3-4, the LDNF and LDCON instructions have been executed. Note that each caused the deferral of an IFA instruction (fetch array element in the E-machine) in QS. Also, for each deferred instruction, a QS segment descriptor was pushed to VS. The LDCON instruction allocated space and made a copy of the descriptor array for the constant which was in the function segment; the new DA is named T1. The VBASE for the constant is 200, the same as the FBASE of the function.

The TAKE operation (3-5, 6) is evaluated by the DM using beating. The descriptor array T2 was created for the result, and was derived from the DA for N by the transformations listed in Chapter III, Appendix A. It is easy to see that this DA is in fact the correct one. Also note that T1 is no longer needed, and has been erased. At this point, VS contains a segment descriptor which points to the QS segment describing the result of the computation to data, which is the evaluation of the subexpression (2,¯2)↑*N* .

Examples 3-7 through 3-9 show the next LDNF instruction and the evaluation of the reversal operation by beating. The process in this case is similar to that for the TAKE. The ADD operation is deferred in 3-10 because both of its operands were array values. The LINK field of the ADD in QS is 2, referring to the operand 2 elements earlier in QS. The top of VS now contains a descriptor for the entire subexpression in QS which has been evaluated at this point. The LDJ instruction (3-11) is executed similarly to LDNF and LDCON in that it defers a value in QS.

The TRANS instruction takes the transpose of thc entire expression which has been dragged along so far. In this case, since its operand is a sum, the transpose is applied to both terms. Notice that although the deferred code in QS has not been altered (3-12), the DA's which it references have been (3-13). The LDN R instruction pushes a value with tag NPT to VS (3-14) as the next instruction is an ASGN (3-15). This instruction notes that R was undefined (see NT, in Example 3-1) and allocates space for its DA and its value array. The space is allocated based on the knowledge of the size of the result deferred in QS. In 3-15, we see the deferral of the assignment. The POP instruction in QS disposes of the value after it has been assigned (in deferring ASGNV, no POPS are used). In 3-16, the state of memory shows the new DA for R; also note that the address of the DA for R (@R) has been entered in NT by the ASGN evaluation.

c. Other Operators (Executed Directly)

The "other operators" include all those APL primitives which cannot be deferred conveniently, or which are evaluated immediately in the D-machine. BASE is in this class because it has a scalar result, while REP, GDU, GDD are included because they require rather complex calculations involving their entire operands simultaneously, which are impossible or difficult to do element-by-element. URHO is easily done by the D-machine, and so is not deferred, as is UIOTA, which produces a J-vector as result. The catenation operator, with operand K, is a direction to catenate the top K elements of VS to form a vector. This is donc immcdiatcly (with tho result boing put in temporary spaoe). The remainder of the operators in this class are dealt with differently, depending on the values of their operands.

```
EXAMPLE 3 - DRAG-ALONG AND BEATING
--------------------------------------------------------------------------------
MEMORY DUMP

ADDR CONTENTS             ADDR CONTENTS             NT: TAG CONTENTS
----+-------------------  ----+-------------------  ---+---+---------------
@M    RC=1      LEN=C5    @N    RC=1      LEN=05    F   FT   @F
 +01  VB=VM     AB=0C0     +01  VB=VN     AB=000    M   DT   @M
 +02      RANK=2           +02      RANK=2          N   DT   @N
 +03  R(1)=0C2 D(1)=02     +03  R(1)=003 D(1)=04    R   DT    0
 +04  R(2)=002 D(2)=01     +04  R(2)=004 D(2)=01

EXAMPLE 3-1: MEMORY BEFORE EXECUTING EXAMPLE CODE
```

```
--------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 0      IORG = 1      FREG = 00000      FBASE = C0200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 050 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0250   IN M

     TAG   VALUE                          OP    VALUE                 LINK  AUX
VS:+-----+--------------------+  QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*    | --> |
-->|

EXAMPLE 3-2: REGISTERS BEFORE EXECUTING EXAMPLE CODE
```

```
--------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 0      IORG = 1      FREG = 00000      FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 054 | C00 | 10C | 0 | 0 | 1 | 0 | C0 |
--> |

    EFFECTIVE ADDR = 0254   IN M

     TAG   VALUE                          OP    VALUE                 LINK  AUX
VS:+-----+--------------------+  QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*    |  00 | IFA | @N                 |     | 0011 | AA
   | SGT | SCODE(SEG.A,1)     |  01 | IFA | @T1                |     | 0001 | 88
   | SGT | SCODE(SEG.B,1)     | --> |
-->|

     LDNF  PUSHED QS(0;) AND VS(1;)
     LDCON PUSHED QS(1;) AND VS(2;)

EXAMPLE 3-3: AFTER  LDNF AND LDCON
```

```
EXAMPLE 3 - DRAG-ALONG AND BEATING
--------------------------------------------------------------------------------
MEMORY DUMP

ADDR CONTENTS             ADDR CONTENTS             ADDR CONTENTS
----+-------------------  ----+-------------------  ----+-------------------
@M    RC=1      LEN=05    @N    RC=2      LEN=05    @T1   RC=1      LEN=04
 +01  VB=VM     AB=000     +01  VB=VN     AB=000     +01  VB=200    AB=094
 +02      RANK=2           +02      RANK=1           +02      RANK=1
 +03  R(1)=C02 D(1)=02     +03  R(1)=003 D(1)=04     +03  R(1)=002 D(1)=01
 +04  R(2)=002 D(2)=01     +04  R(2)=004 D(2)=01

     DA FOR N NOW HAS REFCO OF 2.  T1 IS A COPY OF THE DA FOR THE VECTOR 2,-2

EXAMPLE 3-4: MEMORY AFTER LDCON
```

```
--------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 0      IORG = 1      FREG = 00000      FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 054 | 000 | 100 | 0 | 0 | 1 | 0 | 0C |
--> |

    EFFECTIVE ADDR = 0254   IN M

     TAG   VALUE                          OP    VALUE                 LINK  AUX
VS:+-----+--------------------+  QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*    |  00 | IFA | @T2                |     | 0011 | AA
   | SGT | SCODE(SEG.A,1)     | --> |
-->|

     THE  TAKE  HAS ALTERED THE DA FOR  N, CREATING A NEW COPY.

EXAMPLE 3-5: REGISTERS AFTER TAKE OPERATOR
```

```
--------------------------------------------------------------------------------
MEMORY DUMP

ADDR CONTENTS             ADDR CONTENTS             ADDR CONTENTS
----+-------------------  ----+-------------------  ----+-------------------
@M    RC=1      LEN=05    @N    RC=1      LEN=05    @T2   RC=1      LEN=05
 +01  VB=VM     AB=000     +01  VB=VN     AB=000     +01  VB=VN     AB=002
 +02      RANK=2           +02      RANK=2           +02      RANK=2
 +03  R(1)=002 D(1)=02     +03  R(1)=003 D(1)=04     +03  R(1)=002 D(1)=04
 +04  R(2)=002 D(2)=01     +04  R(2)=004 D(2)=01     +04  R(2)=002 D(2)=01

     THE NEW DA AT  @T2  CONTAINS THE STORAGE ACCESS FUNCTION FOR THE
     TAKE OPERATION ON  N, WHICH WAS PRODUCED BY BEATING.  NOTE IN PARTICULAR
     THAT THE VBASE OF  T2  IS  VN, WHICH POINTS TO THE VALUE ARRAY OF  N, AND
     THAT THE DIMENSION OF  T2  IS  2,2  , AS SPECIFIED BY THE TAKE OPERATOR.
     THE ABASE HAS CHANGED FROM 0 TO 2, TO ACCOUNT FOR THE -2 ELEMENT IN THE
     PARAMETER (I.E. TAKE FROM THE END).  FINALLY, NOTE THAT THE VALUE OF DEL
     IN  T2  IS THE SAME AS THAT FOR  N.

EXAMPLE 3-6: MEMORY AFTER TAKE OPERATOR
```

EXAMPLE 3 - DRAG-ALONG AND BEATING

```
REGISTER DUMP
NEWIT = 0      IORG = 1      FREG = 0000G      FBASE = 0020C

         REL   ORG   LEN  D/E IS  FN  NWT  QP
LS:  +-----+-----+-----+---+---+---+---+----+
     | 056 | 000 | 100 | 0 | 0 | 1 | C | CO |
-->  |

     EFFECTIVE ADDR = 0256   IN M

       TAG   VALUE                        OP    VALUE                  LINK  AUX
VS:+-----+-------------------+   QS:+----+---------------------+-----+------+
   | FMT | *FN MARK FOR F*   |   00 | IFA | @T2                 |     | CO:1 | AA
   | SGT | SCODE(SEG.A,1)    .   01 | IFA | @M                  |     | CO:1 | EB
   | SGT | SCODE(SEG.B,1)    i -->  |
-->|
```

EXAMPLE 3-7: AFTER LDNF M

```
REGISTER DUMP
NEWIT = 0      IURG = 1      FREG = 0C0OG      FBASE = 002C0

         REL   ORG   LEN  D/E IS  FN  NWT  QP
LS:  +-----+-----+-----+---+---+---+---+----+
     | 058 | 000 | 100 | 0 | 0 | 1 | 0 1 00 |
-->  |

     EFFECTIVE ADDR = 0258   IN M

       TAG   VALUE                        OP    VALUE                  LINK  AUX
VS:+-----+-------------------+   QS:+----+---------------------+-----+------+
   | FMT | *FN MARK FOR F*   |   CO | IFA | @T2                 |     | 0C1. | AA
   | SGT | SCODE(SEG.A,1)    |   01 | IFA | @T3                 |     | 0C11 | 33
   | SGT | SCODE(SEG.B,1)    | -->  |
-->|
```

EXAMPLE 3-8: AFTER REV

EXAMPLE 3 - DRAG-ALONG AND BEATING

```
MEMORY DUMP

ADDR CONTENTS                ADDR CONTENTS                ADDR CONTENTS
----+------------------      ----+------------------      ----+------------------
@M    RC=1       LEN=05      @N    RC=1       LEN=05      @T2   RC=1       LEN=05
 +01  VB=VM      AB=000       +01  VB=VN      AB=000       +01  VB=VN      AB=002
 +02     RANK=2               +02     RANK=2               +02     RANK=2
 +03  R(1)=002 D(1)=02        +03  R(1)=003 D(1)=04        +03  R(1)=002 D(1)=04
 +04  R(2)=002 D(2)=01        +04  R(2)=004 D(2)=01        +04  R(2)=002 D(2)=01

                                                          @T3   RC=1       LEN=05
                                                           +01  VB=VM      AB=002
                                                           +02     RANK=2
                                                           +03  R(1)=002 D(1)=-2
                                                           +04  R(2)=0C2 D(2)=01
```

NOTICE THE NEW DA, @T3 , WHICH CONTAINS THE ACCESS FUNCTION FOR THE REVERSAL ON M . THE PARTS WHICH HAVE CHANGED FROM THE DA AT @M ARE ABASE, WHICH IS NOW 2, AND DEL(1), WHICH IS -2 INSTEAD OF 2. THESE CHANGES ACCOUNT FOR THE REVERSAL OF M , ANALOGOUSLY TO THE WAY THE DA AT @T2 ACCOUNTS FOR THE TAKE OPERATION ON N .

EXAMPLE 3-9: AFTER REV

```
REGISTER DUMP
NEWIT = 0      IORG = 1      FREG = 00C00      FBASE = 00200

         REL   ORG   LEN  D/E IS  FN  NWT  QP
LS:  +-----+-----+-----+---+---+---+---+----+
     | 059 | 00C | 1CC | 0 | 0 | 1 | 0 | 0C |
-->  |

     EFFECTIVE ADDR = C259   IN M

       TAG   VALUE                        OP    VALUE                  LINK  AUX
VS:+-----+-------------------+   QS:+----+---------------------+-----+------+
   | FMT | *FN MARK FOR F*   |   00 | IFA | @T2               . |     | 0011 | C_
   | SGT | SCODE(SEG.C,1)    |   01 | IFA | @T3                 |     | 0011 |
-->|                             02 | OP  | ADD                 | 02  | 0011 | _C
                                -->  |
```

EXAMPLE 3-10: AFTER ADD

EXAMPLE 3 - DRAG-ALONG ANC BEATING

```
REGISTER DUMP
NEWIT = 0     IORG = 1     FREG = 00000     FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 061 | 000 | 10C | 0 | 0 | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0261    IN M

     TAG   VALUE                             OP    VALUE                  LINK  AUX
VS:+-----+-------------------+    QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*   |    00 | IFA | @T2                |     | 0011 | C_
   | SGT | SCODE(SEG.C,1)    |    01 | IFA | @T3                |     | 0011 |
   | SGT | SCODE(SEG.D,1)    |    02 | OP  | ADD                | 02  | 0011 | _C
-->|                              03 | IJ  | JCODE(2,1,1)       |     | 0001 | DD
                                 --> |
```

EXAMPLE 3-11: AFTER LDJ

```
REGISTER DUMP
NEWIT = 0     IORG = 1     FREG = 00G00     FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 062 | 000 | 10C | 0 | 0 | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0262    IN M

     TAG   VALUE                             OP    VALUE                  LINK  AUX
VS:+-----+-------------------+    QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*   |    00 | IFA | @T2                |     | 0011 | C_
   | SGT | SCODE(SEG.C,1)    |    01 | IFA | @T3                |     | 0011 |
-->|                              02 | OP  | ADD                | 02  | 0011 | _C
                                 --> |
```

EXAMPLE 3-12: REGISTERS AFTER TRANS

EXAMPLE 3 - DRAG-ALCNG AND BEATING

```
MEMORY DUMP

ADDR CONTENTS                ADDR CONTENTS                ADDR CONTENTS
----+------------------      ----+------------------      ----+------------------
@M    RC=1      LEN=05       @N    RC=1      LEN=05       @T2   RC=1      LEN=05
 +01  VB=VM     AB=000        +01  VB=VN     AB=000        +01  VB=VN     AB=002
 +02     RANK=2               +02     RANK=2               +02     RANK=2
 +03  R(1)=002 D(1)=02        +03  R(1)=003 D(1)=04        +03  R(1)=002 D(1)=01
 +04  R(2)=002 D(2)=01        +04  R(2)=004 D(2)=01        +04  R(2)=002 D(2)=04

                                                          @T3   RC=1      LEN=05
                                                           +01  VB=VM     AB=002
                                                           +02     RANK=2
                                                           +03  R(1)=002 D(1)=01
                                                           +04  R(2)=002 D(2)=-2
```

THE EFFECT OF THE TRANSPUSE WAS TO ALTER THE DA'S AT @T2 AND @T3. THE CHANGE IN BOTH CASES WAS TO INTERCHANGE R(1) WITH R(2), AND D(1) WITH D(2). IT SHOULU BE INTUITIVELY CLEAR THAT THESE DA'S WILL NOW ACCESS THE TRANSPOSES OF THEIR PREVIOUS VALUES.

EXAMPLE 3-13: MEMORY AFTER TRANS (NOTE ALTERED DA'S)

```
REGISTER DUMP
NEWIT = 0     IORG = 1     FREG = 00000     FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 064 | 000 | 10C | 0 | C | 1 | 0 | 00 |
--> |

    EFFECTIVE ADDR = 0264    IN M

     TAG   VALUE                             OP    VALUE                  LINK  AUX
VS:+-----+-------------------+    QS:+-----+--------------------+-----+------+
   | FMT | *FN MARK FOR F*   |    00 | IFA | @T2                |     | 0011 | C_
   | SGT | SCODE(SEG.C,1)    |    01 | IFA | @T3                |     | 0011 |
   | NPT | R                 |    02 | OP  | ADD                | 02  | 0011 | _C
-->|                             --> |
```

EXAMPLE 3-14: AFTER LDN R

```
     EXAMPLE 3 - DRAG-ALONG AND BEATING
--------------------------------------------------------------------------------
     REGISTER DUMP
     NEWIT = 0      IORG = 1      FREG = 00000      FBASE = 00200

           REL    ORG    LEN  D/E ES  FN  NWT  QP
     LS: +------+------+------+---+---+---+---+----+
         | 065 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
     --> |

         EFFECTIVE ADDR = 0265    IN M

          TAG    VALUE                                  OP     VALUE                     LINK  AUX
     VS:+-----+--------------------+   QS:+-----+--------------------+-----+------+
        | FMT | *FN MARK FOR F*    |   00 | IFA | @T2                |     | 0011 | E_
        | SGT | SCODE(SEG.E,1)     |   01 | IFA | @T3                |     | 0011 |
     -->|                              02 | OP  | ADD                | 02  | 0011 |
                                       03 | IFA | @R                 |     | 0011 |
                                       04 | OP  | ASGN               | 02  | 0011 |
                                       05 | POP | 0                  |     | 0011 | _E
                                      --> |

     EXAMPLE 3-15: REGISTERS AFTER ASGN

--------------------------------------------------------------------------------
     MEMORY DUMP

     ADDR CONTENTS                 ADDR CONTENTS                 NT: TAG CONTENTS
     ----+------------------       ----+------------------       ---+---+-----------------
     @M    RC=1       LEN=05       @T2   RC=1       LEN=05       F   FT   @F
      +01  VB=VM      AB=000        +01  VB=VN      AB=002       M   DT   @M
      +02     RANK=2                +02     RANK=2               N   DT   @N
      +03  R(1)=002 D(1)=02         +03  R(1)=002 D(1)=01        R   DT   @R
      +04  R(2)=002 D(2)=01         +04  R(2)=002 D(2)=04

     @N    RC=1       LEN=05       @T3   RC=1       LEN=05
      +01  VB=VN      AB=000        +01  VB=VM      AB=002
      +02     RANK=2                +02     RANK=2
      +03  R(1)=003 D(1)=04         +03  R(1)=002 D(1)=01
      +04  R(2)=004 D(2)=01         +04  R(2)=002 D(2)=-2

     @R    RC=1       LEN=05
      +01  VB=VR      AB=000
      +02     RANK=2
      +03  R(1)=002 D(1)=02
      +04  R(2)=002 D(2)=01

     EXAMPLE 3-16: MEMORY AFTER ASGN
```

RAV and DRHO are difficult to defer in general because of the complex calculations necessary to access an arbitrary element of the result. However, there are special cases which are easy to defer, as follows:

(i) The right operand is a scalar or single-element quantity. The RAV of such a value is a J-vector if it is an integer, or at worst is an explicit one-element vector. Similarly, the DRHO of such a value is deferred in QS as follows:

```
S      value   0   0
IRD    T1      0   R
```

where @T1 is a DA for the result and R is the encoding of the rank. The IRD instruction is essentially a note to the D-machine that the result has dimension described in T1.

(ii) The right operand B is an expression deferred in the form of (i) above. In this case, all that has to be done is change the descriptor array @T1.

(iii) The right operand is of the form

```
IFA   @W   0   R
```

and @W points to a DA which has not been altered by any select operations which upset the ordering of the value part. That is, if *W* is the array specified by @W and *D* is the vector containing the value part, then *W*[;/*L*]↔*D*[(ρ*C*)⊥*L*] for all appropriate values of *L*. In this case, RAV is evaluated by providing a new DA with rank 1 and dimension ×/ρ*W* . DRHO can be deferred if ×/ρ*A* , where *A* is the left operand of the DRHO, is less than or equal to ×/ρ*C* also by providing a new DA with dimension *A*.

If none of the above apply, then RAV and DRHO are evaluated immediately by creating temporary values in M.

d. Other Operators and Compound Operators (Deferrable)

The D-machine evaluates this subclass of operator instructions by deferring E-machine code in QS. The expansions are detailed in Appendix C and should be easy to understand with a knowledge of the way the E-machine works. We will here discuss only the SUBS instruction and the compound operators, as their behavior is somewhat more complex.

The SUBS K operation corresponds to the symbol [in an APL program. When decoded, it expects the top of VS to contain a QS segment descriptor for a rank-K quantity and the next K entries on VS to be either scalars or QS segment descriptors for the subscript expressions. An empty subscript position is created by the LDSEG instruction with its operand a segment descriptor SCODE(0, 0, 0) of length 0.

There are two important cases to consider:

(i) If the subscriptee is beatable, then the subscript expressions are examined in turn, starting from the rightmost (deepest in VS) to find scalars or J-vectors. If found for, say, the I$\underline{\text{th}}$ coordinate, the equivalent of INX I with that operand is performed on the subscriptee by beating, causing new DA's to be created for it. The VS entry for this subscript is then deleted if it was a scalar. If it was a J-vector, then the VS entry is changed to the empty segment and the QS entry is deleted by moving all of QS down 1 to fill in the space (with appropriate adjustments to descriptors). If, after all subscripts have been examined it is found that the remaining stacked subscripts are either empty or non-existent, then the result already exists, in standard form, in QS. In this case, the remaining empty segment descriptors are removed from VS and the result is the QS descriptor at the top of VS. Otherwise, the remaining subscripts are treated as in the second case, described in the next paragraph.

(ii) If there are explicit non-scalar or non-J-vector subscript expressions and/or the subscriptee is not beatable, then the subscripts must be dragged along in QS. This is done by creating temporary index accumulators (opcode XT) in QS and generating E-machine code to activate the necessary subscript evaluations at the right times. If the subscriptee is a reduction, QS is transformed according to the transformation $(OP/A)[\mathscr{L}] \longrightarrow OP/A[\mathscr{L};]$ and evaluation continues as above. The details of the subscript expansion are shown in Appendix C. Example 4 illustrates the process which has just been described.

In evaluating a GDF, the machine first examines the operands. If they contain deferred operators, then they are evaluated to temporary space first. This is done to avoid unnecessary recalculation of subexpressions necessary to compute a GDF. It also guarantees the possibility of applying SF transforms to GDF expressions by beating. Then all that is necessary is to alter the access masks in the AUX fields of the deferred left operand in QS to provide the proper access method for the E-machine. This is illustrated in Example 5 below. If the GDF reduces to a simple case, e.g., if one of the operands is a scalar, then the expression is treated as a normal scalar operator expression (see part a above).

Efficient evaluation of reductions along coordinate K of the reducee R (in the E-machine) depend on transformation TR11 (see Chapter II) which allows permutation of the reduction coordinate by transposing the reducee. In evaluating a REDUCE along coordinate K the reducee is first checked to see if it fits into one of the special cases of reduction:

(i) Empty reduction coordinate. The result is then an array with value ((K≠ιρρR)/ρR)ρIDENT where R is the reducee and IDENT is the identity element for the reduction operator.

(ii) Reduction coordinate of length 1. The result is then R[[K] IORG] If reducee is a scalar, the result is R.

(iii) Reducee is a vector. In this case, the reduction is activated immediately in the E-machine, since the result is a scalar.

If none of the special cases is satisfied, the reduction is deferred by first doing the transpose of TR11 if necessary, and generating the deferred code in QS as shown in Appendix C.

EXAMPLE 4: SUBSCRIPTING IN D-MACHINE

Consider the APL expression A[ι4;;2;V] where A is a rank-4 array with ρA↔5,4,6,3 and V↔3,2,1,2 , with the index origin IORG ↔ 1. The D-machine for evaluating this expression is

```
250    LDNF     V                Vector V
252    LDS      2                Scalar 2
254    LDSEG    SCODE(0, 0, 0)   Empty subscript
256    LDS      4                Scalar 4
258    UIOTA                     Gives ι4
259    LDNF     A                Array A
261    SUBS     4                Do the subscript, expected operand rank is 4
263    ....
```

The following memory and register dumps show the steps the D-machine goes through to evaluate this expression.

EXAMPLE 4 - SUBSCRIPTING IN D-MACHINE

```
MEMORY DUMP

ADDR CONTENTS                ADDR CONTENTS                NT: TAG CONTENTS
----+------------------      ----+------------------      ---+---+-----------------
@A    RC=1      LEN=07       @V    RC=1      LEN=04       A   DT   @A
 +01  VB=V1     AB=0C0        +01  VB=VV     AB=000       V   DT   @V
 +02     RANK=4               +02     RANK=1
 +03  R(1)=005 D(1)=72        +03  R(1)=004 D(1)=01
 +04  R(2)=004.D(2)=18
 +05  R(3)=006 D(3)=03
 +06  R(4)=003 D(4)=01
```

EXAMPLE 4-1: MEMORY BEFORE EXECUTING EXAMPLE CODE

```
REGISTER DUMP
NEWIT = 0     IORG = 1      FREG = 00C00      FBASE = 0020C

       REL   ORG   LEN  D/E IS  FN  NWT  QP
LS:  +-----+-----+-----+---+---+---+---+-----+
     | 061 | C00 | 10C | 0 | 0 | 1 | C | 00 |
-->  |

     EFFECTIVE ADDR = 0261   IN M

      TAG   VALUE                            OP    VALUE                  LINK  AUX
VS:+-----+-------------------+   QS:+-----+----------------------+----+------+
   | ..  |       ...         |   00 | IFA | @V                   |    | 0001 | AA
   | SGT | SCODE(SEG.A,1)    |   01 | IJ  | JCODE(4,1,0)         |    | 0001 | BB
   | ST  | 2                 |   02 | IFA | @A                   |    | 1111 | CC
   | SGT | SCODE(SEG.N(L,0)  |  --> |
   | SGT | SCODE(SEG.B,1)    |
   | SGT | SCODE(SEG.C,1)    |
-->|
```

VS CONTENTS ARE THE SUBSCRIPTS AND SUBSCRIPTEE. NOTE THE ACCESS MASKS IN THE AUX FIELD OF QS. THEY INDICATE THAT V AND THE J-VECTOR ARE VECTORS, AND A IS A RANK-4 ARRAY.

EXAMPLE 4-2: AFTER ALL BUT THE SUBS OPERATOR

EXAMPLE 4 - SUBSCRIPTING IN D-MACHINE

```
REGISTER DUMP
NEWIT = 0     IORG = 1      FREG = 00C00      FBASE = 00200

       REL   ORG   LEN  D/E IS  FN  NWT  QP
LS:  +-----+-----+-----+---+---+---+---+-----+
     | 063 | 000 | 10C | 0 | 0 | 1 | 0 | 0C |
-->  |

     EFFECTIVE ADDR = 0263   IN M

      TAG   VALUE                            OP    VALUE                  LINK  AUX
VS:+-----+-------------------+   QS:+-----+----------------------+----+------+
   | ..  |       ...         |   00 | JMP | 0                    | 06 |      | D_
   | SGT | SCODE(SEG.D,1)    |   01 | IFA | @V                   |    | 0001 | EE
-->|                             02 | IFA | @T1                  |    | 0111 | FF
                                 03 | XT  | XCODE(0,3,1)         | 03 |      |
                                 04 | XT  | XCODE(0,3,1)         |    |      |
                                 05 | XT  | XCODE(0,2,1)         |    |      |
                                 06 | IXL | 0                    |    | 0100 |
                                 07 | XS  | 0                    | 04 |      |
                                 08 | IXL | 0                    |    | 0010 |
                                 09 | XS  | 0                    | 05 |      |
                                 10 | ISC | SCODE(SEG.E,1)       |    | 0001 |
                                 11 | XS  | 0                    | 06 |      |
                                 12 | SG  | SCODE(SEG.F,1)       | 09 |      |
                                 13 | IRD | @T2                  |    | 0111 | _D
                                --> |
```

VS AND QS HAVE BEEN TRANSFORMED BY THE SUBS OPERATION. THE SCALAR SUBSCRIPT REDUCED THE RANK OF A BY 1, AND THE INTERVAL VECTOR SHORTENED THE FIRST COORDINATE (SEE DA AT @T1). THE REST OF THE CODE GENERATED IN QS IS FOR CALCULATING EXPLICIT SUBSCRIPT VALUES, WHICH ARE KEPT IN THE XT ENTRIES. THESE ENTRIES CONSTITUTE A PSEUDO-ITERATION STACK. (SEE SECTION E)

EXAMPLE 4-3: REGISTERS AFTER SUBS

```
MEMORY DUMP

ADDR CONTENTS                ADDR CONTENTS                ADDR CONTENTS
----+------------------      ----+------------------      ----+------------------
@A    RC=1      LEN=07       @V    RC=2      LEN=04       @T2   RC=1      LEN=06
 +01  VB=V1     AB=0C0        +01  VB=VV     AB=00C        +01  VB=       AB=C00
 +02     RANK=4               +02     RANK=1               +02     RANK=3
 +03  R(1)=005 D(1)=72        +03  R(1)=004 D(1)=01        +03  R(1)=004 D(1)=16
 +04  R(2)=004 D(2)=18                                     +04  R(2)=004 D(2)=04
 +05  R(3)=006 D(3)=03       @T1   RC=1      LEN=06        +05  R(3)=004 D(3)=01
 +06  R(4)=003 D(4)=01        +01  VB=VA     AB=003
                              +02     RANK=3
                              +03  R(1)=004 D(1)=72
                              +04  R(2)=004 D(2)=18
                              +C5  R(3)=003 D(3)=01
```

EXAMPLE 4-4: MEMORY AFTER SUBS

EXAMPLE 5: GDF IN D-MACHINE

In the example expression, *M*∘.×*N*, both *M* and *N* are matrices with ρ*M*↔4,3 and *N*↔ρ3,2. D-machine code for this expression is

```
250     LDNF      N
252     LDNF      M
254     GDF       MUL       Do GDF
256     ...
```

Examples 5-1,2 show the machine state before evaluating this code. In 5-3, the GDF operation has been deferred in QS. Notice that the access mask of M in the AUX field of QS has been changed. The IRD entry, whose operand DA gives the dimension of the result, contains 1111 in its AUX field, which instructs the EM to use a 4-level iteration stack to evaluate the expression. The 1100 AUX for M says that M-indices come from the two highest iterations, while the 0011 AUX for N indicates that N is to use the two lowest.

An equivalent formulation of the contents of QS at this point is that it represents the GDF in the form:

```
for I := 0 step 1 until 3 do
    for J := 0 step 1 until 2 do
        for K := 0 step 1 until 2 do
            for L := 0 step 1 until 1 do
                RESULT [I;J;K;L] := M[I;J] ×N[K;L];
    EXAMPLE 5 - GDF IN D-MACHINE
--------------------------------------------------------------------------------
    REGISTER DUMP
    NEWIT = 0      IORG = 1      FREG = 00000      FBASE = 00200

          REL    ORG    LEN  C/E IS  FN  NWT  QP
    LS:  +-----+------+------+---+---+---+---+-----+
         | 054 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
    --> |

         EFFECTIVE ADDR = 0254    IN M

          TAG    VALUE                           OP     VALUE                    LINK  AUX
    VS:+-----+--------------------+  QS:+-----+----------------------+-----+------+
       | ..  |        ...         |  00 | IFA | @N                   |     | 0011 | AA
       | SGT | SCODE(SEG.A,1)     |  01 | IFA | @M                   |     | 0011 | BB
       | SGT | SCODE(SEG.B,1)     | --> |
    -->|


    EXAMPLE 5-1: REGISTERS BEFORE GDF
--------------------------------------------------------------------------------
    MEMORY DUMP

    ADDR CONTENTS                ADDR CONTENTS
    ----+-----------------       ----+-----------------
    @M    RC=1      LEN=05       @N    RC=1      LEN=05
     +01  VB=VM     AB=000        +01  VB=VN     AB=000
     +02     RANK=2               +02     RANK=2
     +03  R(1)=004 D(1)=03        +03  R(1)=003 D(1)=02
     +04  R(2)=003 D(2)=01        +04  R(2)=002 D(2)=01


    EXAMPLE 5-2: MEMORY BEFORE GDF
```

```
    EXAMPLE 5 - GDF IN D-MACHINE
--------------------------------------------------------------------------------
    REGISTER DUMP
    NEWIT = 0      IORG = 1      FREG = 00000      FBASE = 00200

          REL    ORG    LEN  D/E IS  FN  NWT  QP
    LS:  +-----+------+------+---+---+---+---+-----+
         | 056 | 000 | 100 | 0 | 0 | 1 | 0 | 00 |
    --> |

         EFFECTIVE ADDR = 0256    IN M

          TAG    VALUE                           OP     VALUE                    LINK  AUX
    VS:+-----+--------------------+  QS:+-----+----------------------+-----+------+
       | ..  |        ...         |  00 | IFA | @N                   |     | 0011 | C_
       | SGT | SCODE(SEG.C,1)     |  01 | IFA | @M                   |     | 1100 |
    -->|                             02 | GOP | MUL                  |     | 1111 |
                                     03 | IRD | @T1                  |     | 1111 | _C
                                    --> |


    EXAMPLE 5-3: AFTER GDF - NOTE CHANGED AUX FIELDS IN QS
--------------------------------------------------------------------------------
    MEMORY DUMP

    ADDR CONTENTS                ADDR CONTENTS                ADDR CONTENTS
    ----+-----------------       ----+-----------------       ----+-----------------
    @M    RC=2      LEN=05       @N    RC=2      LEN=05       @T1   RC=1      LEN=07
     +01  VB=VM     AB=000        +01  VB=VN     AB=000        +01  VB=      AB=000
     +02     RANK=2               +02     RANK=2               +02     RANK=4
     +03  R(1)=004 D(1)=03        +03  R(1)=003 D(1)=02        +03  R(1)=004 D(1)=18
     +04  R(2)=003 D(2)=01        +04  R(2)=002 D(2)=01        +04  R(2)=003 D(2)=06
                                                               +05  R(3)=003 D(3)=02
                                                               +06  R(4)=002 D(4)=01

        @T1  WAS CREATED SIMPLY TO RECORD THE RANK AND DIMENSION VECTOR OF
        THE RESULT OF DOING THE OUTER PRODUCT.  THE OPCODE  IRD  (IN QS(3;))
        SIGNIFIES THAT ITS OPERAND DA IS DESCRIPTIVE, AND IS NOT TO BE
        EXECUTED.  IN THE E-MACHINE, IRD  IS IGNORED.

    EXAMPLE 5-4: MEMORY AFTER GDF
```

E. The E-Machine

The E-machine is a simple stack-oriented computer which evaluates array-valued expressions by iterating element-by-element over their index sets. The EM takes its instructions from the instruction buffer (QS), where they were put by the D-machine. Other machine registers are used in the same way as in the DM.

The central task of the EM is to access individual array elements in computing array-valued expressions. As most of the complexity of the E-machine is related to this task, we first discuss the accessing mechanisms in the EM. Given this, it is a simple matter to explain the instruction set of the machine.

1. Array Accessing

a. Indexing Environment

Array reference instructions are entered in QS in the form

```
IFA    @VAR    0    MASK
```

where @VAR is the address of a DA in M, and MASK is a logical access mask. When such an instruction is first entered in QS by the D-machine, it is done without regard to its context in the input expression. The E-machine must, in order to evaluate it, determine its context, which takes the form of an indexing environment for an array reference. The indexing environment of an instruction in QS is determined by how the segment containing the instruction was activated, which in turn relates to the form of the original expression input to the D-machine.

(i) If the QP field of the top of LS is zero, then the environment is simple, and array references within this segment are based directly on the iteration stack. A simple environment arises in variables not affected by explicit subscripting or which are not operands in expressions which cause expansions to be made by the DM. For example, in the statement $A \leftarrow B+C$, all variables have simple environment.

(ii) If the QP field of LS is non-zero, then the environment is complex, and array references in this segment are controlled by a pseudo-iteration stack. In the statement A←B+C[V;W] , A and B will have simple environments, but C will be complex as the reference to C is embedded in a segment resulting from the expansion of the subscript operator. Note that this concept is recursive. For example, we can also say that the environment of the subexpression C[V;W] is simple. This recursiveness allows arbitrary levels of subscript nesting to be handled by the drag-along scheme of the D-machine.

The segment containing the IFA @C instruction is activated in the EM by an SG instruction referring to a sequence of entries in QS of the form:

```
XT    XCODE(a, m1, c1)
XT    XCODE(b, m2, c2) .
```

Here, a and b are indices for C calculated from the subscripts V and W by the expanded subscript code in QS. These quantities are, in turn, computed from the current values in IS. m1 and m2 are the maximum permissible values of a and b derived from ρC, and c1 and c2 are change flags. Thus, these XT entries correspond to the CNT, MAX, and CH fields of the iteration stack, and are therefore called a pseudo-iteration stack (pseudo-IS).

b. Initialization of Access Instructions

Each array accessing instruction must be bound to its indexing environment when first executed. This process is described below for IFA instructions and is analogous for IA and IJ.

(i) Determine index sources

The encoded access mask in the AUX field of an instruction is used to determine its indexing environment. For example, if the environment is simple and the bit pattern in AUX is 0101 and the IS is four deep, then the index sources are determined by (0, 1, 0, 1)/0, 1, 2, 3 which is the vector 1, 3. Call this vector INX. Had the QP field of LS indicated a complex indexing environment, then INX would have been based on the length of the pseudo-IS rather than on the length of IS.

(ii) Set up iteration control block

An <u>iteration control block</u> (ICB) is established at the top of QS, containing the coefficients of the storage mapping function from the DA for the array (DEL) and the INX vector, calculated above. An ICB contains one word for each coordinate of the array being accessed, as shown below. The fields marked Q1 and Q2 are both encoded into the VALUE field of QS using the function QCODE (see Appendix A). The contents of the I$^{\underline{th}}$ ICB entry are:

field	contents
OP	<u>if</u> simple environment <u>then</u> NT <u>else</u> QT
LINK	INX [I]
AUX	0
Q2	DEL [I]
Q1	if simple environment then DEL [I] × (MAX field of IS entry selected by LINK field) else 0

In addition, the last entry in an ICB is given opcode NLT or QLT, depending on its environment.

(iii) Initialize QS entry

The Q1 fields of the ICB just established are added to the ABASE found in the array's descriptor array to produce the sum S. VBASE is also fetched from the DA, and the DA is "erased" from QS by subtracting 1 from its reference count. The original IFA entry is then replaced by

FA   QCODE(VBASE, S)   IPTR   0

where IPTR is a pointer to the beginning of the ICB for this array.

This completes the initialization of array references. In effect, what has been done is to replace the context-independent reference created by the D-machine, by information which binds the reference to its indexing environment, and which contains all information necessary to access the array (in the ICB).

c. The Index Unit

The index unit (IU) is invoked by the E-machine every time it executes an array-access instruction that has been initialized as above (i.e., FA, A, J). Using the information in the instruction, its ICB,and IS or a pseudo-IS, the IU accesses the appropriate array element and pushes it to VS. The IU functions differently, depending on the indexing environment:

(i) Simple environment

In this case, we know a priori that the elements of the array will be accessed in a simple order, determined by the way IS is cycled, and this information can be used to minimize the re-computation of the storage mapping function for each element of the array. The IU looks at the iteration stack entries for this array (specified in the ICB), starting at the right-most coordinate. If the IS entry has changed (noted by CH bit) but not recycled, then the IS adds the DEL component from the ICB to S; if there was a change and a recycle, the Q1 field is subtracted from S.

The new S value is stored back in the instruction. This process continues until an IS entry with no changes is found, in which case none of the higher IS entries contain changes either. If the iteration is going backwards, as in a reduce, then addition and subtraction are interchanged.

(ii) Complex environment

In the complex case, there is no way of predicting in advance how the indices will proceed and each change requires an explicit evaluation of part of the mapping function. This is done similarly to the simple case, by examining the pseudo-IS for each coordinate of the array. If a change is recorded (in the X3 part of the XT entry) then the new index (X1 part) is multiplied by DEL. This result is added to S and the Q1 field of the ICB is subtracted from S with the new S stored back in QS. Finally, the product just found is stored in the Q1 part of the ICB. This field thus records partial values of the mapping polynomial.

The behavior of the machine in array accessing, as described above, is illustrated in Example 6.

2. Instruction Set

Instructions in the E-machine can be considered in three groups:

a. Simple instructions

b. Control instructions

c. Micro-instructions, used primarily for maintaining pseudo-iteration stacks.

In addition, as seen in the previous section, the instructions buffer contains entries for pseudo-iteration stacks (opcode XT) and iteration control blocks (NT, QT, NLT, QLT). Table 3 summarizes the E-machine repertoire, and Appendix B contains a detailed algorithmic description of the E-machine's behavior. The remainder of this section discusses these instructions in both functional and "programming" terms.

a. Simple instructions

The S instruction, Load Scalar, pushes its value to VS with tag ST. IFA fetches an array element according to its operand DA and the indexing environment, and pushes it to VS with tag ST; similarly, IJ pushes an element of a J-vector to VS, while IA pushes an address of an array element (tag AT). These instructions can be considered simply at the programming level, as just described, although the mechanism which they invoke is much more complex, as was seen in the previous section.

The instructions OP and GOP have as operands the names of arithmetic functions in the EM (monadic or dyadic). Executing an OP or GOP invokes the named function, which operates on the top of VS, deleting the operands and pushing the result, with tag ST. (This process is illustrated in Example 1.) NIL is a No-op, and does nothing. Recall from Section D and Appendix C that IRD and IRP are generated by the D-machine to keep track of intermediate results in doing drag-along. As they have no use in the E-machine, they are changed to NIL when first executed.

b. Control instructions

The main control instructions are SGV and SG, whose operands are QS segment descriptors. SGV pushes this descriptor to VS (with tag SGT) and is thus analogous to LDSEG in the DM. SG activates the named segment by pushing an entry to LS; in this instruction, the LINK field is significant, in that it can change the indexing environment, JMP, J0, J1, JN0, and JN1 are simply relative jumps within QS; RED is also a relative jump, but in addition, it pushes to VS an entry with tag RT, to be used as an accumulator for a reduction. (RED is generated by the DM only in conjunction with reductions.)

MIT is used primarily to activate reduction segments. It takes ST entries from the top of VS and uses them to push new iterations to IS. When the MIT execution reaches an SGT entry on the top of VS, the referenced segment is activated by pushing the descriptor information to LS. (See Appendix C for a description of how reduction segments are deferred in QS.)

c. Micro-instructions

The set of micro-instructions are used by the E-machine to maintain pseudo-iteration stacks in QS. They result from D-machine expansions of subscripting and related operations. The micro-instructions are fully explained in Table 3-C, and the DM expansions in Appendix C illustrate their use.

TABLE 3

E-Machine Instruction Set

Notes:

a. Each instruction is in the form

OP VALUE LINK AUX.

In the discussion, K is the address of the instruction in QS.

b. Instructions starting with the letter "I" are "uninitialized." That is, they have not yet been bound to their indexing environments. They are changed to similar instructions without the leading "I" when first executed.

TABLE 3-A

E-Machine — Simple Instructions

Operation	Name	Definition
S	Load Scalar	Push VALUE to VS, with tag ST.
IFA FA	Load Array Element	IFA causes initialization, as described in Section E.1.B., and the instruction becomes FA. FA fetches an array element determined by the indexing environment and pushes the value to VS with tag ST.
IA A	Load Array Address	IA causes initialization and the instruction becomes A. A is similar to FA except that the (encoded) address of the selected element is pushed to VS with tag AT.
IJ J	Load J-Vector Element	IJ is similar to IFA, and becomes J after initialization. The VALUE field is an encoded descriptor of a J-vector, the correct element of which is computed and pushed to VS with tag ST.
OP GOP	Scalar Operator	The VALUE field is the name of a scalar arithmetic operator. This is invoked and takes its operands from the top of VS, leaving a result there after deleting the operands.
NIL	No Operation	No operation.
IRD IRP	Result Dimension	These instructions are used by the D-machine and are left in QS when a segment is turned over to the E-machine. Since they are of no use to the EM, they are changed to NIL the first time encountered.

TABLE 3-B

E-Machine — Control Instructions

Operation	Name	Definition
SGV	Load Segment Descriptor	The VALUE field is a QS segment descriptor, with addresses relative to K. Make these addresses absolute and push the descriptor to VS with tag SGT.
SG	Activate Segment	The VALUE field is as in SGV, and LINK, if nonzero, points to a pseudo-iteration stack in QS. Activate the segment by pushing an entry to LS, using the LINK information to alter the QP field of LS if necessary.
JMP J0 J1 JN0 JN1	Jump Jump if 0 Jump if 1 Jump if 0 nondestructive Jump if 1 nondestructive	Potential jump destination is K+LINK, where LINK is considered as a signed number. JMP is unconditional. The others are conditional on the value on top of VS. J0 and J1 also pop VS.
RED	Begin Reduction	Push an element with tag RT to VS to act as a reduction accumulator, and jump to K+LINK.
MIT	Mark and Iterate	Scalar values on top of VS are used to start a new iteration nest in IS. The absolute value of the VS value, less 1, is the MAX field in IS; the iteration direction (DIR) is forward (0) if VS is positive, otherwise backward (1). The CNT field of IS is initialized to 0 or MAX, depending on whether DIR is 0 or 1. Moreover, the first entry in IS has its MRK bit set to 1; all others are 0. Each VS value is popped. Finally, when an SGT entry is found it is popped and the named segment is activated in LS.

TABLE 3-C

E-Machine — Micro-Instructions

Operation	Name	Definition
POP	Pop	Pop top element of VS.
DUP	Duplicate	Fetch the VS entry, LINK elements from top of VS, and push it to VS. (Does not disturb original copy.)
ORG	Load IORG	Push current value of IORG register to VS (tag ST).
CY	Cycle	Step IS and repeat the current segment if IS hasn't overflowed.
LVE	Leave	De-activate the current segment, erasing any associated IS entries.
RPT	Repeat	Repeat current segment from beginning. (Does not affect IS.)
CAS	Case	If top of VS is not an integer scalar, then error else if the value is N, then pop VS and execute the instruction at K+N and resume execution at K+LINK.
VXC	Exchange	Interchange top two entries on VS.
LX1 LX2	Load from Pseudo-IS	LINK fields are relative pointers to XT entries. Push X1 (or X2) field of referenced entry to VS, tag ST.
SX1 SX2	Store in Pseudo-IS	Store top (ST) entry on VS in X1 (or X2) field of referenced XT entry. Pop VS.
IXL XL	Index load	IXL is initialized to give XL, in which the LINK field points to IS or a pseudo-IS element. XL gets the current iteration value, adds IORG, and pushes the result to VS with tag ST.
XS	Index Store	Subtract IORG from ST entry on top of VS; store in X1 field of XT entry at K-LINK in QS; if the value just stored is negative or greater than the X2 field of the same word, signal an error. Set the X3 field (change bit) to 1, and pop VS.
XC	Index Change	Set the change bit (X3 field) of the referenced XT entry to 1.
ISC SC	Activate Segment Conditional	ISC is initialized to SC in same way as IXL. The VALUE field of the instruction is a QS segment descriptor. If the change bit in the referenced IS or pseudo-IS entry is 1, then the segment is activated. Otherwise, the change bit of the XT entry referenced by the following instruction is set to 0, and this instruction is skipped.

EXAMPLE 6:

This example illustrates typical behavior of the E-machine. Consider the APL statement

```
E[I;]←EP>|¯1+(+/(1 2 2 ⍉PT∘.-PT[I;])*2)*0.5
```

and suppose it is encountered by the machine when the variables are as below:

EP is 0.0001

I is 2

```
PT is 0  0         E is 0  1  1  0
      0  1              1  0  0  1
      1  0              0  0  0  0
      1  1              0  0  0  0
```

The D-machine code for this statement is as follows:

D-Machine Code for Statement in Example 6:

```
Addr   Op       Operand            Comments

200    LDS      0.5
202    LDS      2
204    LDSEG    SCODE(0,0,0)       Empty subscript
206    LDNF     I
208    LDNF     PT
210    SUBS     2                  Result is PT[I;]
212    LDNF     PT
214    GDF      SUB                PT∘.-PT[I;]
216    LDCON    50                 Constant vector 1,2,2
218    TRANS                       1 2 2 ⍉PT∘.-PT[I;]
219    PWR                         (1 2 2 ⍉PT∘.-PT[I;])*2
220    RED      1  ADD             +/(1 2 2 ⍉PT∘.-PT[I;])*2
223    PWR                         (+/(1 2 2 ⍉PT∘.-PT[I;])*2)*0.5  (Call this R )
224    LDS      -1
226    ADD                         ¯1+R
227    MOD                         |¯1+R
228    LDNF     EP
230    GT                          EP>|¯1+R
231    LDSEG    SCODE(0,0,0)       Empty subscript
233    LDNF     I
235    LDN      E
237    SUBS     2                  E[I;]
239    ASGN                        E[I;]←EP>|¯1+R
240    ...
250    RC=1     LEN=4              Header for DA of constant 1,2,2
251    VB=0     AB=54              Rest of DA
252        RANK=1
253    R(1)=3   D(1)=1
254    RC=1     LEN=4              Header for value of constant 1,2,2
255    1
256    2                           Value array
257    3
```

Example 6-1 shows the instruction buffer containing the deferred code to evaluate the sample statement. The transpose operation was evaluated in the D-machine using beating, and its results are manifested in the access masks (AUX field) in the instructions at locations 3 and 4.

Four temporary descriptor arrays were created by the DM as follows:

@T1 DA for *PT*[2;] . (Recall that I is 2 in this example.)

@T2 DA containing dimension of the result of the GOP operation, in this case 4,2.

@T3 DA containing dimension of the reduction result, in this case 4.

@T4 DA for *E*[2;]

The deferred code is equivalent to the following:

```
for J := 0 step 1 until 3 do
    begin
        REDUCE := 0;
        for K := 1 step -1 until 0 do
            REDUCE := REDUCE + (PT[J;K] PT[2;K])*2;
        E[2;J] := 0.0001>|¯1+(REDUCE*0.5);
    end
```

The remainder of the example shows the D-machine's progress through the code in QS, and contains comments which explain the machine's actions at each step.

```
EXAMPLE 6 -- E-MACHINE
--------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IURG = 0      FREG = 00C00      FBASE = C0200   ISMK = 00

      REL   ORG   LEN  D/E IS  FN  NWT  UP            CTR   MAX  DIR CH  MRK
_S: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 04C | C00 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 000 | 000 | 022 | 1 | 1 | 0 | 3 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0000   IN QS

     TAG   VALUE                               OP    VALUE                LINK  AUX
VS: +-----+--------------------+    QS: +-----+---------------------+-----+------+
    | FMT | FCODE(-1,0,F)      |    00  | S   | 0.5                 |     |      |
--> |                               C1  | RED | 0                   | 08  |      |
                                    02  | S   | 2                   |     |      |  A_
                                    03  | IFA | ∂T1                 |     | 0001 |
                                    04  | IFA | ∂PT                 |     | 0011 |
                                    05  | GUP | SUB                 | 02  | 0011 |
                                    06  | IRO | ∂T2                 |     | 0011 |
                                    C7  | OP  | PWR                 | 05  | 0011 |
                                    08  | UP  | ADD                 | 07  | 0011 |  _A
                                    09  | SGV | SCODE(SEG.A,1)      |     |      |
                                    10  | S   | -2                  |     |      |
                                    11  | MIT | 0                   |     |      |
                                    12  | IRO | ∂T3                 |     | 00C1 |
                                    13  | OP  | PWR                 | 13  | 0001 |
                                    14  | S   | -1                  |     |      |
                                    15  | OP  | ADD                 | 02  | 00C1 |
                                    16  | OP  | MOD                 |     | 00C1 |
                                    17  | S   | 0.0001              |     |      |
                                    18  | OP  | GT                  | 02  | 0001 |
                                    19  | IA  | ∂T4                 |     | 00C1 |
                                    2C  | OP  | ASGN                | 02  | 00C1 |
                                    21  | POP | 0                   |     |      |
                                    --> |
```

THE D-MACHINE HAS JUST PASSED CONTROL TO THE E-MACHINE. NO EXECUTION HAS TAKEN PLACE YET. THE FUNCTION MARK ON VS WAS PLACED THERE BY ACTIVATING FUNCTION F. THE CONTENTS OF THE MARK ARE THE PREVIOUS VALUES OF FREG (-1) AND IORG (0), AND THE NAME OF THE FUNCTION (F).

SEGMENT A WITHIN QS EVALUATES THE REDUCTION FOUND IN THE SOURCE CODE. THE ITERATION STACK IS SET UP TO DO THE EQUIVALENT OF THE "FOR J := 0 STEP 1 UNTIL 3" ITERATION.

EXAMPLE 6-1: STATE OF THE REGISTERS BEFORE EXECUTION

```
EXAMPLE 6 -- E-MACHINE
--------------------------------------------------------------------------------
MEMORY DUMP

ADDR CONTENTS                  ADDR CONTENTS                 NT: TAG CONTENTS
----+---------------------     ----+--------------------     ---+---+----------------
VPT   RC=2       LEN=09        ∂PT   RC=2      LEN=05        F    FT   ∂F
 +01  0                         +01  VB=VPT    AB=000        I    ST     2
 +02  0                         +02     RANK=2               PT   DT   ∂PT
 +03  0                         +03  R(1)=004 D(1)=02        E    DT   ∂E
 +C4  1                         +04  R(2)=002 D(2)=01        EP   ST     0.0001
 +05  1
 +06  0                        ∂E    RC=1      LEN=05
 +07  1                         +01  VB=VE     AB=000
 +08  1                         +02     RANK=2
                                +03  R(1)=004 D(1)=04
VE    RC=2       LEN=17         +04  R(2)=004 D(2)=01
 +01  0
 +02  1                        ∂T1   RC=1      LEN=04
 +03  1                         +C1  VB=VPT    AB=004
 +04  0                         +02     RANK=1
 +05  1                         +03  R(1)=002 D(1)=01
 +C6  0
 +C7  0                        ∂T2   RC=1      LEN=05
 +08  1                         +01  VB=       AB=000
 +09  0                         +02     RANK=2
 +10  C                         +03  R(1)=004 D(1)=02
 +11  0                         +C4  R(2)=002 D(2)=01
 +12  0
 +13  0                        ∂T3   RC=1      LEN=04
 +14  0                         +01  VB=       AB=000
 +15  0                         +02     RANK=1
 +16  C                         +03  R(1)=004 D(1)=01

                               ∂T4   RC=1      LEN=04
                                +01  VB=VE     AB=008
                                +02     RANK=1
                                +03  R(1)=C04 D(1)=01
```

NOTE THAT IN THE NAMETABLE, THE ENTRY FOR THE IDENTIFIER F POINTS TO ∂F. THE TAG OF THE ENTRY IDENTIFIES IT AS A FUNCTION NAME. ∂F IS THE ADDRESS OF THE FUNCTION DESCRIPTOR FOR F, WHICH IS NOT SHOWN.

EXAMPLE 6-2: STATE OF MEMCRY BEFORE EXECUTION

```
EXAMPLE 6 -- E-MACHINE
---------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200     ISMK = 00

      REL    ORG    LEN   D/E IS   FN   NWT   QP            CTR    MAX   DIR CH  MRK
LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+----+
    | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 000  | 001  | 0  | 1  | .  |
    | 001  | 000  | 022  | 1  | 1  | 0  | 3  | 00  | --> |
--> |

    EFFECTIVE ADDR = 0001    IN QS

     TAG    VALUE                         OP     VALUE                  LINK   AUX
VS:+-----+------------------+  QS:+-----+-------------------------+-----+-------
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
-->|

     THE  S  INSTRUCTION (LOAD SCALAR) PUSHED ITS OPERAND (0.5) TO  VS.

EXAMPLE 6-3: AFTER S
```

```
---------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200     ISMK = 00

      REL    ORG    LEN   D/E IS   FN   NWT   QP            CTR    MAX   DIR CH  MRK
LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+----+
    | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 000  | 001  | 0  | 1  | .  |
    | 011  | 000  | 022  | 1  | 1  | 0  | 3  | 00  | --> |
--> |

    EFFECTIVE ADDR = 0011    IN QS

     TAG    VALUE                         OP     VALUE                  LINK   AUX
VS:+-----+------------------+  QS:+-----+-------------------------+-----+-------
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | RT  | 0                |
   | SGT | SCODE(SEG.A,1)   |
   | ST  | -2               |
-->|

     THE RED OPERATOR PUSHED THE  RT  ENTRY, TO BE USED AS AN ACCUMULATOR
     FOR THE REDUCTION, AND JUMPED TO QS(9 ).  THE  SGV  INSTRUCTION (AT 9)
     PUSHED ITS OPERAND (THE DESCRIPTOR FOR SEGMENT A) TO VS.
     THE  S  INSTRUCTION (AT 10) PUSHED THE  -2  VALUE TO VS.
     THESE TWO ENTRIES WILL BE USED BY THE  MIT  INSTRUCTION TO ACTIVATE
     THE REDUCTION SEGMENT.

EXAMPLE 6-4: AFTER RED, SGV, AND S
```

```
EXAMPLE 6 -- E-MACHINE
---------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200     ISMK = 01

      REL    ORG    LEN   D/E IS   FN   NWT   QP            CTR    MAX   DIR CH  MRK
LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+-------+
    | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 000  | 003  | 0  | 1     1 |
    | 012  | 000  | 022  | 1  | 1  | 0  | 3  | 00  |     | 001  | 001  | 1  | 1     1 |
    | 000  | 002  | 007  | 1  | 1  | 0  | 1  | 00  | --> |
--> |

    EFFECTIVE ADDR = 0002    IN QS

     TAG    VALUE                         OP     VALUE                  LINK   AUX
VS:+-----+------------------+  QS:+-----+-------------------------+-----+-------+
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | RT  | 0                |
-->|

     MIT  USED THE SCALAR  -2  ON TOP OF  VS  TO START A NEW ITERATION.
     THE LENGTH OF THE ITERATION IS 2, AND THUS THE MAX FIELD IN THE ITERATION
     STACK IS SET TO 1.  THE NEGATIVE SIGN OF THE  VS  ENTRY SIGNIFIED THAT THE
     ITERATION IS TO RUN BACKWARDS (DIR=1); HENCE CTR STARTS AT 1 INSTEAD OF 0.
     THE NEXT  VS  ENTRY WAS A SEGMENT DESCRIPTOR FOR SEGMENT  A  IN QS.
     MIT  USED THIS TO ACTIVATE THE SEGMENT, BY PUSHING A NEW ENTRY TO  LS.
     NOTE THAT IN THE NEW  LS  ENTRY, THE  NWT  BIT IS 1; THIS WAS THE PREVIOUS
     VALUE OF  NEWIT.  NEWIT IS NOW 1 BECAUSE A NEW ITERATION HAS BEEN STARTED.

EXAMPLE 6-5: AFTER MIT
```

```
---------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200     ISMK = 01

      REL    ORG    LEN   D/E IS   FN   NWT   QP            CTR    MAX   DIR CH  MRK
LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+----+
    | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 000  | 003  | 0  | 1  | 1  |
    | 012  | 000  | 022  | 1  | 1  | 0  | 3  | 00  |     | 001  | 001  | 1  | 1  | 1  |
    | 001  | 002  | 007  | 1  | 1  | 0  | 1  | 00  | --> |
--> |

    EFFECTIVE ADDR = 0003    IN QS

     TAG    VALUE                         OP     VALUE                  LINK   AUX
VS:+-----+------------------+  QS:+-----+-------------------------+-----+-------+
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | RT  | 0                |
   | ST  | 2                |
-->|

     THE FIRST INSTRUCTION OF THE NEWLY-ACTIVATED SEGMENT (SEG.A) IS   S ,
     AT QS(2;).  THIS INSTRUCTION PUSHED ITS OPERAND (2) TO  VS.

EXAMPLE 6-6: AFTER S (AT QS(2;) )
EXAMPLE 6 -- E-MACHINE
------------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200    ISMK = 01

      REL   ORG   LEN  D/E IS  FN  NWT  QP          CTR   MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 |     | 001 | 001 | 1 | 1 | 1 |
    | 001 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0003   IN QS

     TAG   VALUE                            OP    VALUE                LINK  AUX
VS: +-----+---------------------+    QS: +-----+-------------------+-----+------+
   | FMT | FCODE(-1,0,F)        |    00 | S   | 0.5               |     |      |
   | ST  | 0.5                  |    01 | RED | 0                 | 08  |      |
   | RT  | 0                    |    02 | S   | 2                 |     |      | A_
   | ST  | 2                    |    03 | FA  | QCODE(VPT,4)      | 19  |      |
-->|                                 04 | IFA | @PT               |     | 0011 |
                                     05 | GOP | SUB               | 02  | 0011 |
                                     06 | IRD | @T2               |     | 0011 |
                                     07 | OP  | PWR               | 05  | 0011 |
                                     08 | OP  | ADD               | 07  | 0011 | _A
                                     09 | SGV | SCODE(SEG.A,1)    |     |      |
                                     10 | S   | -2                |     |      |
                                     11 | MIT | 0                 |     |      |
                                     12 | IRD | @T3               |     | 0001 |
                                     13 | OP  | PWR               | 13  | 0001 |
                                     14 | S   | -1                |     |      |
                                     15 | OP  | ADD               | 02  | 0001 |
                                     16 | OP  | MOD               |     | 0001 |
                                     17 | S   | 0.0001            |     |      |
                                     18 | OP  | GT                | 02  | 0001 |
                                     19 | IA  | @T4               |     | 0001 |
                                     20 | OP  | ASGN              | 02  | 0001 |
                                     21 | POP | 0                 |     |      |
                                     22 | NLT | QCODE(1,1)        | 01  |      |
                                    --> |
```

LOCATION 3 IN QS, WHICH PREVIOUSLY CONTAINED AN IFA INSTRUCTION, HAS BEEN INITIALIZED TO FA. THE VALUE FIELD NOW CONTAINS VPT , THE BASE ADDRESS REFERENCED IN THE DA AT @T1, AND THE ABASE (=4) FROM THAT DA. IN ADDITION, THE LINK FIELD OF QS(3;) IS NOW A RELATIVE POINTER TO QS(22;), WHICH IS THE ITERATION CONTROL BLOCK FOR THIS ARRAY. THE SECOND ELEMENT OF THE ICB ENTRY (I.E. THE Q2 FIELD) IS THE DEL FOR THIS ARRAY, TAKEN FROM @T1. (SEE EXAMPLE 6-2 FOR CONTENTS OF T1). THE FIRST ELEMENT (Q1 FIELD) IS DEL TIMES THE MAX VALUE IN THE TOP ENTRY ON IS.

LS HAS NOT CHANGED YET BECAUSE THE NEWLY-CREATED FA INSTRUCTION HAS NOT YET BEEN EXECUTED. THE INITIALIZATION PROCESS ALSO ERASED THE DA STARTING AT @T1, WHICH IS NO LONGER REFERENCED ANYWHERE IN THE MACHINE.

EXAMPLE 6-7: AFTER IFA

```
EXAMPLE 6 -- E-MACHINE
------------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200    ISMK = 01

      REL   ORG   LEN  D/E IS  FN  NWT  QP          CTR   MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 |     | 001 | 001 | 1 | 1 | 1 |
    | 002 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0004   IN QS

     TAG   VALUE                            OP    VALUE                LINK  AUX
VS: +-----+---------------------+    QS: +-----+-------------------+-----+------+
   | FMT | FCODE(-1,0,F)        |    00 | S   | 0.5               |     |      |
   | ST  | 0.5                  |    01 | RED | 0                 | 08  |      |
   | RT  | 0                    |    02 | S   | 2                 |     |      | A_
   | ST  | 2                    |    03 | FA  | QCODE(VPT,5)      | 19  |      |
   | ST  | 0                    |    04 | IFA | @PT               |     | 0011 |
-->|                                 05 | GOP | SUB               | 02  | 0011 |
                                     06 | IRD | @T2               |     | 0011 |
                                     07 | OP  | PWR               | 05  | 0011 |
                                     08 | OP  | ADD               | 07  | 0011 | _A
                                     09 | SGV | SCODE(SEG.A,1)    |     |      |
                                     10 | S   | -2                |     |      |
                                     11 | MIT | 0                 |     |      |
                                     12 | IRD | @T3               |     | 0001 |
                                     13 | OP  | PWR               | 13  | 0001 |
                                     14 | S   | -1                |     |      |
                                     15 | OP  | ADD               | 02  | 0001 |
                                     16 | OP  | MOD               |  .  | 0001 |
                                     17 | S   | 0.0001            |     |      |
                                     18 | OP  | GT                | 02  | 0001 |
                                     19 | IA  | @T4               |     | 0001 |
                                     20 | OP  | ASGN              | 02  | 0001 |
                                     21 | POP | 0                 |     |      |
                                     22 | NLT | QCODE(1,1)        | 01  |      |
                                    --> |
```

THE ADDRESS IN QS(3;) HAS BEEN UPDATED BY THE INDEX UNIT AND THE VALUE IT REFERS TO HAS BEEN PUSHED TO VS. THUS THE VALUE (0) ON TOP OF VS AT THIS POINT IS PT(2;1). (RECALL THAT THE EFFECTIVE ADDRESS OF AN ARRAY ELEMENT REFERENCED IN AN FA INSTRUCTION IS THE SUM OF ITS CODED PARTS, PLUS 1 (TO COMPENSATE FOR THE ARRAY HEADER WORD)).

\ EXAMPLE 6-8: AFTER FA

```
EXAMPLE 6 -- E-MACHINE
------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200    ISMK = 01

       REL    ORG    LEN   D/E IS  FN  NWT  QP               CTR    MAX  DIR CH  MRK
LS:  +-----+-----+-----+---+---+---+---+----+  IS:  +-----+-----+---+---+---+
     | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |       | 000 | 003 | 0 | 1 | 1 |
     | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 |       | 001 | 001 | 1 | 1 | 1 |
     | 003 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | -->   |
-->  |

    EFFECTIVE ADDR = 0005    IN QS

      TAG    VALUE                          OP     VALUE                 LENK  AUX
VS:+-----+-------------------+      QS:+-----+---------------------+----+------+
   | FMT | FCODE(-1,0,F)      |      00 | S   | 0.5                 |    |      |
   | ST  | 0.5                |      01 | RED | 0                   | 08 |      |
   | RT  | 0                  |      02 | S   | 2                   |    |      |  A_
   | ST  | 2                  |      03 | FA  | QCODE(VPT,5)        | 19 |      |
   | ST  | 0                  |      04 | FA  | QCODE(VPT,1)        | 19 |      |
   | ST  | 0                  |      05 | GOP | SUB                 | 02 | 0011 |
-->|                                 06 | IRD | @T2                 |    | 0011 |
                                     07 | OP  | PWR                 | 05 | 0011 |
                                     08 | OP  | ADD                 | 07 | 0011 |  _A
                                     09 | SGV | SCODE(SEG_A,1)      |    |      |
                                     10 | S   | -2                  |    |      |
                                     11 | MIT | 0                   |    |      |
                                     12 | IRD | @T3                 |    | 0001 |
                                     13 | OP  | PWR                 | 13 | 0001 |
                                     14 | S   | -1                  |    |      |
                                     15 | OP  | ADD                 | 02 | 0001 |
                                     16 | OP  | MOD                 |    | 0001 |
                                     17 | S   | 0.0001              |    |      |
                                     18 | OP  | GT                  | 02 | 0001 |
                                     19 | IA  | @T4                 |    | 0001 |
                                     20 | OP  | ASGN                | 02 | 0001 |
                                     21 | POP | 0                   |    |      |
                                     22 | NLT | QCODE(1,1)          | 01 |      |
                                     23 | NT  | QCODE(6,2)          |    |      |
                                     24 | NLT | QCODE(1,1)          | 01 |      |
                                  -->  |
```

THE IFA AT QS[4;] HAS BEEN CHANGED TO FA , AS IN EXAMPLE 6-7, AND THE FA HAS BEEN EXECUTED, AS IN 6-8. THE TOP TWO ELEMENTS ON VS ARE NOW PT[2;1] AND PT[0;1]. ALSO NOTE THE TWO NEW ENTRIES ON THE TOP OF QS , WHICH ARE THE ICB FOR THE FA AT QS[4;].

EXAMPLE 6-9: AFTER QS[4;] (INITIALIZATION AND EXECUTION)

```
EXAMPLE 6 -- E-MACHINE
------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200    ISMK = 01

       REL    ORG    LEN   D/E IS  FN  NWT  QP               CTR    MAX  DIR CH  MRK
LS:  +-----+-----+-----+---+---+---+---+----+  IS:  +-----+-----+---+---+---+
     | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |       | 000 | 003 | 0 | 1 | 1 |
     | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 |       | 001 | 001 | 1 | 1 | 1 |
     | 005 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | -->   |
-->  |

    EFFECTIVE ADDR = 000'    IN QS

      TAG    VALUE                          OP     VALUE                 LINK  AUX
VS:+-----+-------------------+      QS:+-----+---------------------+----+------+
   | FMT | FCODE(-1,0,F)      |      00 | S   | 0.5                 |    |      |
   | ST  | 0.5                |      01 | RED | 0                   | 08 |      |
   | RT  | 0                  |      02 | S   | 2                   |    |      |  A_
   | ST  | 2                  |      03 | FA  | QCODE(VPT,5)        | 19 |      |
   | ST  | 0                  |      04 | FA  | QCODE(VPT,1)        | 19 |      |
-->|                                 05 | GOP | SUB                 | 02 | 0011 |
                                     06 | NIL | 0                   |    |      |
                                     07 | OP  | PWR                 | 05 | 0011 |
                                     08 | OP  | ADD                 | 07 | 0011 |  _A
                                     09 | SGV | SCODE(SEG.A,1)      |    |      |
                                     10 | S   | -2                  |    |      |
                                     11 | MIT | 0                   |    |      |
                                     12 | IRD | @T3                 |    | 0001 |
                                     13 | OP  | PWR                 | 13 | 0001 |
                                     14 | S   | -1                  |    |      |
                                     15 | OP  | ADD                 | 02 | 0001 |
                                     16 | OP  | MOD                 |    | 0001 |
                                     17 | S   | 0.0001              |    |      |
                                     18 | OP  | GT                  | 02 | 0001 |
                                     19 | IA  | @T4                 |    | 0001 |
                                     20 | OP  | ASGN                | 02 | 0001 |
                                     21 | POP | 0                   |    |      |
                                     22 | NLT | QCODE(1,1)          | 01 |      |
                                     23 | NT  | QCODE(6,2)          |    |      |
                                     24 | NLT | QCODE(1,1)          | 01 |      |
                                  -->  |
```

THE SUB HAS BEEN DONE. (IN THE E-MACHINE, GOP IS TREATED SAME AS OP.) THE IRD OPERATION DECREASES THE REFCO OF ITS OPERAND BY 1 AND REPLACES ITSELF BY NIL, THE NO-OP, BECAUSE IRD IS USED BY THE D-MACHINE BUT NOT BY THE E-MACHINE.

EXAMPLE 6-10: AFTER SUB,IRD

```
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200   ISMK = 01

      REL    ORG    LEN   D/E IS  FN  NWT  QP            CTR    MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | C0 |     | 001 | 001 | 1 | 1 | 1 |
    | 006 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0008   IN QS

     TAG   VALUE                            OP     VALUE                LINK  AUX
VS:+-----+------------------+  QS:+-----+-------------------+-----+------+
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | RT  | 0                |
   | ST  | 0                |
-->|
```

PWR (AT QS(7;1)) WAS APPLIED TO THE TOP 2 ELEMENTS ON THE VALUE STACK, 0 AND 2 : THESE OPERANDS WERE DELETED AND THE RESULT OF THE OPERATION HAS BEEN PUSHED TO VS. (0 * 2 = 0)

EXAMPLE 6-11: AFTER PWR

```
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 0020C   ISMK = 01

      REL    ORG    LEN   D/E IS  FN  NWT  QP            CTR    MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | C0 |     | 001 | 001 | 1 | 1 | 1 |
    | 007 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | --> |
--> |

    EFFECTIVE ADDR = C009   IN QS

     TAG   VALUE                            OP     VALUE                LINK  AUX
VS:+-----+------------------+  QS:+-----+-------------------+-----+------+
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | ST  | 0                |
-->|
```

THE ADD OPERATION, SEEING THAT ITS SECOND OPERAND HAS TAG RT, GIVES AS ITS RESULT THE FIRST OPERAND, WITH TAG ST. THIS IS ACCORDING TO THE DEFINITION OF REDUCTION.

EXAMPLE 6-12: AFTER ADD

```
REGISTER DUMP
NEWIT = 0      IORG = 0      FREG = 00000      FBASE = 00200   ISMK = 01

      REL    ORG    LEN   D/E IS  FN  NWT  QP            CTR    MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 |     | 000 | 001 | 1 | 1 | 1 |
    | 000 | 002 | 007 | 1 | 1 | 0 | 1 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0002   IN QS

     TAG   VALUE                            OP     VALUE                LINK  AUX
VS:+-----+------------------+  QS:+-----+-------------------+-----+------+
   | FMT | FCODE(-1,0,F)    |              ***QS UNCHANGED***
   | ST  | 0.5              |
   | ST  | 0                |
-->|
```

IN THE LAST FRAME, THE SEGMENT WAS COMPLETED, SINCE ITS RELATIVE ADDRESS WAS THE SAME AS ITS LENGTH. HOWEVER, SINCE THE IS BIT WAS SET FOR THAT SEGMENT, THE IS WAS STEPPED BUT DIDN'T OVERFLOW. THUS, LS WAS RE-INITIALIZED TO THE BEGINNING OF THE SEGMENT, TO BE REPEATED WITH THE NEW IS VALUES. NOTE THAT NEWIT NOW IS 0. AT THIS POINT, THE EQUIVALENT OF THE ALGOLIC "REDUCE := REDUCE + ..." HAS BEEN DONE FOR J=0 AND K=1.

THE SECOND PASS THROUGH THE REDUCTION SEGMENT PROCEEDS SIMILARLY TO THE FIRST, EXCEPT THAT NO FURTHER INITIALIZATIONS NEED BE DONE. AT THE END OF THIS ITERATION, REL=LEN IN LS AND, AS BEFORE, THE ITERATION STACK WILL BE STEPPED. HOWEVER, THIS TIME IT OVERFLOWS, SO BOTH LS AND IS ARE POPPED, RETURNING THE MACHINE TO THE MAIN SEGMENT. (SEE NEXT FIGURE)

EXAMPLE 6-13: BEGINNING OF SEGMENT WITH STEPPED IS

```
EXAMPLE 6 -- E-MACHINE
--------------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 0020C     ISMK = 00

       REL   ORG   LEN  D/E IS  FN  NWT  OP            CTR   MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 012 | 000 | 022 | 1 | 1 | 0 | 3 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0012   IN QS

     TAG   VALUE                          OP    VALUE                  LINK  AUX
VS:+-----+------------------+   QS:+-----+--------------------+----+------+
   | FMT | FCODE(-1,0,F)    |   00 | S   | 0.5                |    |      |
   | ST  | 0.5              |   01 | RED | 0                  | 08 |      |
   | ST  | 1                |   02 | S   | 2                  |    |      | A_
-->|                            03 | FA  | QCODE(VPT,4)       | 19 |      |
                                04 | FA  | QCODE(VPT,0)       | 19 |      |
                                05 | GOP | SUB                | 02 | 0011 |
                                06 | NIL | 0                  |    |      |
                                07 | OP  | PWR                | 05 | 0011 |
                                08 | OP  | ADD                | 07 | 0011 | _A
                                09 | SGV | SCODE(SEG.A,1)     |    |      |
                                10 | S   | -2                 |    |      |
                                11 | MIT | 0                  |    |      |
                                12 | IAD | @T3                |    | 0001 |
                                13 | OP  | PWR                | 13 | 0001 |
                                14 | S   | -1                 |    |      |
                                15 | OP  | ADD                | 02 | 0001 |
                                16 | OP  | MOD                |    | 0001 |
                                17 | S   | 0.0001             |    |      |
                                18 | OP  | GT                 | 02 | 0001 |
                                19 | IA  | @T4                |    | 0001 |
                                20 | OP  | ASGN               | 02 | 0001 |
                                21 | POP | 0                  |    |      |
                                22 | NLT | QCODE(1,1)         | 01 |      |
                                23 | NT  | QCODE(6,2)         |    |      |
                                24 | NLT | QCODE(1,1)         | 01 |      |
                               --> |

    REDUCE SEGMENT IS DONE.  ITS RESULT (1) IS ON TOP OF VS.
    NOTE THAT NEWIT WAS RESTORED TO 1 WHEN LS WAS POPPED.

    THIS STAGE CORRESPONDS TO THE COMPLETION OF THE "FOR K" LOOP WITH J=0.

EXAMPLE 6-14: AFTER RETURN FROM REDUCTION
```

```
EXAMPLE 6 -- E-MACHINE
--------------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 1      IORG = 0      FREG = 00000      FBASE = 00200     ISMK = 00

       REL   ORG   LEN  D/E IS  FN  NWT  OP            CTR   MAX  DIR CH  MRK
LS: +-----+-----+-----+---+---+---+---+----+ IS: +-----+-----+---+---+---+
    | 040 | 000 | 075 | 0 | 0 | 1 | 3 | 00 |     | 000 | 003 | 0 | 1 | 1 |
    | 020 | 000 | 022 | 1 | 1 | 0 | 3 | 00 | --> |
--> |

    EFFECTIVE ADDR = 0020   IN QS

     TAG   VALUE                          OP    VALUE                  LINK  AUX
VS:+-----+------------------+   QS:+-----+--------------------+----+------+
   | FMT | FCODE(-1,0,F)    |   00 | S   | 0.5                |    |      |
   | ST  | 1                |   01 | RED | 0                  | 08 |      |
   | AT  | QCODE(VE,8)      |   02 | S   | 2                  |    |      | A_
-->|                            03 | FA  | QCODE(VPT,4)       | 19 |      |
                                04 | FA  | QCODE(VPT,0)       | 19 |      |
                                05 | GOP | SUB                | 02 | 0011 |
                                06 | NIL | 0                  |    |      |
                                07 | OP  | PWR                | 05 | 0011 |
                                08 | OP  | ADD                | 07 | 0011 | _A
                                09 | SGV | SCODE(SEG.A,1)     |    |      |
                                10 | S   | -2                 |    |      |
                                11 | MIT | 0                  |    |      |
                                12 | NIL | 0                  |    |      |
                                13 | OP  | PWR                | 13 | 0001 |
                                14 | S   | -1                 |    |      |
                                15 | OP  | ADD                | 02 | 0001 |
                                16 | OP  | MOD                |    | 0001 |
                                17 | S   | 0.0001             |    |      |
                                18 | OP  | GT                 | 02 | 0001 |
                                19 | A   | QCODE(VE,8)        | 06 |      |
                                20 | OP  | ASGN               | 02 | 0001 |
                                21 | POP | 0                  |    |      |
                                22 | NLT | QCODE(1,1)         | 01 |      |
                                23 | NT  | QCODE(6,2)         |    |      |
                                24 | NLT | QCODE(1,1)         | 01 |      |
                                25 | NLT | QCODE(3,1)         |    |      |
                               --> |

    QS(12:0) THROUGH QS(18:1) HAVE BEEN EXECUTED.  NOTE THAT THE IA AT QS(19:1)
    WAS TRANSFORMED TO  A   AND THAT ITS RESULT IS THE CODED ADDRESS WITH
    TAG 'AT' ON TOP OF VS.

EXAMPLE 6-15: BEFORE ASGN
```

```
      EXAMPLE 6 -- E-MACHINE
    -----------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 1       IORG = 0       FREG = 00000       FBASE = 00200    ISMK = 00

            REL    ORG    LEN   D/E IS   FN   NWT   QP             CTR    MAX   DIR CH   MRK
      LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+----+
          | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 000  | 003  | 0  | 1  | 1  |
          | 022  | 000  | 022  | 1  | 1  | 0  | 3  | 00  | --> |
      --> |

          EFFECTIVE ADDR = 0022    IN QS

           TAG    VALUE                           OP     VALUE                   LINK   AUX
      VS:+------+-------------------+    QS:+------+----------------------+-----+-------+
         | FMT  | FCODE(-1,0,F)     |                 ***QS UNCHANGED***
      -->|

           AFTER ASGN AND VPOP.  THE VALUE CN VS HAS BEEN STURED AT VE+1+8 IN MEMURY.
           SINCE THE SEGMENT HAS BEEN COMPLETEU, THE    IS    WILL BE STEPPED AND
           LS WILL BE RESET TO THE BEGINNING SINCE THERE IS NO OVERFLOW.
           THIS STAGE CORRESPONDS TO ONE PASS THROUGH THE "FOR J" RANGE, WITH J=0.
```

EXAMPLE 6-16: AT END OF MAIN SEGMENT, FIRST TIME THROUGH

```
    -----------------------------------------------------------------------------
      MEMORY DUMP

      ADDR CONTENTS                ADDR CONTENTS                ADDR CONTENTS
      ----+------------------      ----+------------------      ----+------------------
      @PT    RC=1      LEN=05      VPT    RC=1      LEN=09      VE     RC=1      LEN=17
       +01   VB=VPT    AB=000       +01   0                      +01   0
       +02      RANK=2              +02   0                      +02   1
       +03   R(1)=004 D(1)=02       +03   0                      +03   1
       +04   R(2)=002 D(2)=01       +04   1                      +04   0
                                    +05   1                      +05   1
      @E     RC=1      LEN=05       +06   0                      +06   0
       +01   VB=VE     AB=000       +07   1                      +07   0
       +02      RANK=2              +08   1                      +08   1
       +03   R(1)=004 D(1)=04                                    +09   1
       +04   R(2)=004 D(2)=01                                    +10   0
                                                                 +11   0
                                                                 +12   0
                                                                 +13   0
                                                                 +14   0
                                                                 +15   0
                                                                 +16   0

           ENTRIES FOR @T1,...,@T4 NOW HAVE REFCOS OF 0, AND HAVE BEEN ADDED TO THE
           LINKED AVAILABILITY LIST, ALTHOUGH THIS IS NOT SHOWN HERE.
           THE ENTRY IN THE VALUE ARRAY FOR  E , AT  VE+9  IN MEMORY, HAS BEEN
           CHANGED TO 1 BY THE ASGN OPERATION.  THIS ENTRY IS  E(2;0).
```

EXAMPLE 6-17: STATE OF M AFTER FIRST TIME THROUGH THE SEGMENT

```
      EXAMPLE 6 -- E-MACHINE
    -----------------------------------------------------------------------------
      REGISTER DUMP
      NEWIT = 0       IORG = 0       FREG = 00000       FBASE = 00200    ISMK = 00

            REL    ORG    LEN   D/E IS   FN   NWT   QP             CTR    MAX   DIR CH   MRK
      LS: +------+------+------+----+----+----+----+-----+ IS: +------+------+----+----+----+
          | 040  | 000  | 075  | 0  | 0  | 1  | 3  | 00  |     | 003  | 003  | 0  | 1  | 1  |
          | 022  | 000  | 022  | 1  | 1  | 0  | 3  | 00  | --> |
      --> |

          EFFECTIVE ADDR = 0022    IN QS

           TAG    VALUE                           OP     VALUE                   LINK   AUX
      VS:+------+-------------------+    QS:+------+----------------------+-----+-------+
         | FMT  | FCODE(-1,0,F)     |    00 | S    | 0.5                  |     |       |
      -->|                               01 | RED  | 0                    | 08  |       |
                                         02 | S    | 2                    |     |       |  A_
                                         03 | FA   | QCODE(VPT,4)         | 19  |       |
                                         04 | FA   | QCODE(VPT,6)         | 19  |       |
                                         05 | GOP  | SUB                  | 02  | 0011  |
                                         06 | NIL  | 0                    |     |       |
                                         07 | OP   | PWR                  | 05  | 0011  |
                                         08 | OP   | ADD                  | 07  | 0011  |  _A
                                         09 | SGV  | SCODE(SEG.A,1)       |     |       |
                                         10 | S    | -2                   |     |       |
                                         11 | MIT  | 0                    |     |       |
                                         12 | NIL  | 0                    |     |       |
                                         13 | OP   | PWR                  | 13  | 0001  |
                                         14 | S    | -1                   |     |       |
                                         15 | OP   | ADD                  | 02  | 0001  |
                                         16 | OP   | MOD                  |     | 0001  |
                                         17 | S    | 0.0001               |     |       |
                                         18 | OP   | GT                   | 02  | 0001  |
                                         19 | A    | QCODE(VE,11)         | 06  |       |
                                         20 | OP   | ASGN                 | 02  | 0001  |
                                         21 | POP  | 0                    |     |       |
                                         22 | NLT  | QCODE(1,1)           | 01  |       |
                                         23 | NT   | QCODE(6,2)           |     |       |
                                         24 | NLT  | QCODE(1,1)           | 01  |       |
                                         25 | NLT  | QCODE(3,1)           |     |       |
                                        --> |

           THE MAIN SEGMENT WAS REPEATED 3 MORE TIMES IN THE SAME WAY AS SHOWN
           FOR THE FIRST PASS.  AT THIS POINT, 3 MORE VALUES HAVE BEEN STORED
           AND THE    IS    ENTRY CORRESPONDING TO THIS SEGMENT HAS BEEN EXHAUSTED.
           THIS POINT CORRESPONDS TO THE COMPLETION OF  "FOR J".
```

EXAMPLE 6-18: REGISTERS AFTER NEXT THREE PASSES THROUGH SEGMENT

```
EXAMPLE 6 -- E-MACHINE
--------------------------------------------------------------------------------
REGISTER DUMP
NEWIT = 3      IORG = 0      FREG = 00000     FBASE = 00200

      REL   ORG   LEN  D/E IS  FN  NWT  QP
LS: +-----+-----+-----+---+---+---+---+----+
    | 040 | C00 | 075 | C | 0 | 1 | 3 | 00 |
--> |

    EFFECTIVE ADDR = 0240   IN M

     TAG   VALUE                      OP    VALUE              LINK  AUX
VS:+-----+------------------+  QS:+-----+-----------------+----+------+
   | FMT | FCODE(-1,0,F)    | --> |
-->|
```

THE LAST FIGURE WAS THE END OF THE SEGMENT. THUS, IS WAS STEPPED. SINCE IT OVERFLOWED, IS AND LS WERE POPPED. DE-ACTIVATING THAT SEGMENT CHANGED CONTROL FROM THE E- TO THE D-MACHINE AND THEREFORE QI WAS RESET TO THE BEGINNING OF THE SEGMENT JUST COMPLETED.

EXAMPLE 6-19: REGISTERS AT COMPLETICN OF E-MACHINE EVALUATION.

```
--------------------------------------------------------------------------------
MEMORY DUMP

ADDR CONTENTS            ADDR CONTENTS            ADDR CONTENTS
----+------------------  ----+------------------  ----+------------------
@PT   RC=1     LEN=05    VPT   RC=1     LEN=09    VE     RC=1     LEN=17
 +01  VB=VPT   AB=000     +01  0                   +01   0
 +02     RANK=2           +02  0                   +02   1
 +03  R(1)=004 D(1)=02    +03  0                   +03   1
 +04  R(2)=002 D(2)=01    +04  1                   +04   0
                          +05  1                   +05   1
@E    RC=1     LEN=05     +06  0                   +06   0
 +01  VB=VE    AB=000     +07  1                   +07   0
 +02     RANK=2           +08  1                   +08   1
 +03  R(1)=004 D(1)=04                             +09   1
 +04  R(2)=004 D(2)=01                             +10   0
                                                   +11   0
                                                   +12   1
                                                   +13   0
                                                   +14   0
                                                   +15   0
                                                   +16   0
```

NOTICE THAT THE VALUES AT VE+9,10,11,12 HAVE CHANGED FROM EXAMPLE 6-2. THESE CORRESPOND TO E(2;), THE ENTIRE ROW OF E TO BE CALCULATED.

EXAMPLE 6-20: MEMORY AT COMPLETION OF E-MACHINE EVALUATION

APPENDIX A

SUMMARY OF REGISTERS, ENCODINGS AND TAGS

This appendix summarizes the uses of all machine registers and details the fields in the various stacks. In addition, the several encodings used as parametric functions in the design description are outlined. Because of the parametric nature of the design, not much will be said about field sizes except to indicate the range of the contents of a particular field or register. We assume that in any particular incarnation of such a machine, all the fields are "big enough" to contain their contents. In the detailed algorithms of Appendix B, the registers are construed as arrays of scalars with some kind of encoding imposed upon the contents, if necessary. While not completely rigorous, this approach serves to show how the machine works without having to explicitly encode and decode all references to registers at each step.

A. Registers

1. LS (Location Counter Stack)

Field Name	Column Index	Contents
REL	0	Relative location in segment. Generally points to the next instruction to be fetched.
ORG	1	Segment origin. For D-machine segments, this is relative to FBASE. In the E-machine, the effective address is +/*LS*[*LI*-1;0,1] and in the D-machine it is *FBASE*++/*LS*[*LI*-1;0,1].
LEN	2	Length of segment. For D-machine segments, this is in words, and for the E-machine, this is the number of QS entries for the segment.
D/E	3	Segment mode. This field is 0 for the D-machine and 1 for E-machine segments.
IS	4	Iteration mark. Has value 1 if this segment is associated with an iteration in IS; otherwise it is 0.

FN	5	Function mark. Has value 1 (else 0) if this is the main segment of an active function.
NWT	6	NEWIT value, stacked when a new iteration is activated.
QP	7	QS pointer. Used by index unit for expression indexed from QS rather than IS. (See Section E.)

2. IS (Iteration Control Stack)

Field Name	Column Index	Contents
CTR	0	Current iteration count. This value is always non-negative and varies between 0 and the value in the MAX field, in the direction indicated by the DIR field.
MAX	1	Maximum iteration count.
DIR	2	Direction of count. (0 for positive, 1 for negative.) If positive, then CTR is initialized to 0;otherwise it is initialized to MAX.
CH	3	Change. Used by STEPIS routine in main control cycle to mark all IS entries which have changed since the last cycle.
MRK	4	Mark. Has value 1 for the outermost iteration of each nest. Otherwise, it is 0. (See ISMK register, below.)

3. VS (Value Stack)

Field Name	Column Index	Contents
TAG	0	Tag field. Identifies kind of entry in value field.
VALUE	1	Value.

4. QS (Instruction Buffer)

Field Name	Column Index	Contents
OP	0	E-machine operation code. The QS contains instructions deferred by the D-machine for later execution by the E-machine. Occasionally this field will contain a tag, such as XT, for an entry which is a temporary value for the EM rather than an executable instruction.
VALUE	1	Value. Contains the value in immediate instructions and the operand for others.

LINK	2	Link. This is a signed integer used to reference other instructions and entries in QS. It is taken relative to the QS index of the entry in which it is found. By keeping links and segment origins relative in QS, all deferred code is relocatable.
AUX	3	Access mask. Contains an encoding (MCODE) of the iteration indices to use in accessing an array expression.

5. NT (Nametable)

Field Name	Column Index	Contents
INX	0	Symbol index. Since NT is content-addressable, the value of INX must be carried with each entry. These indices (or names) may be assigned in any arbitrary way. There is no built-in restriction on their use.
TAG	1	Tag. Same as tag field in VS.
CONTENTS	2	Value. Same as in VS.

6. M(Memory)

In the APL machine, M is considered to be a vector of length MLENGTH of words which can be addressed between BOTM and TOPM. The particular encodings used in M are not specified except as necessary, e.g., in instructions such as LDSEG, the M-entry containing the operand is in SCODE encoding. Otherwise, each scalar value is assumed to take up one machine word, as is each instruction. This is clearly inefficient in space utilization, and it would be expected that any real implementation would specify more reasonable and detailed encodings for various kinds of values. Nothing in the machine design is based on the word as the primary unit of memory in the machine, so there should be no problem in making such modifications.

7. Other Scalar-Valued Registers

Register Name	Contents
LI	LS index. (All stack indices point to the next available entry in the stack.)
II	IS index.
VI	VS index.
QI	QS index.
NI	NT index.
BOTP TOPP	POOL pointers for M allocation.
ARRAVAIL DAAVAIL	Pointers to beginning of availability chains for M allocation.
FREG	VS index of innermost active function mark. When a function is activated, the previous values of FREG and IORG are stacked in VS in the function mark, and restored on return.
IORG	Index origin for innermost active function.
FBASE	Function origin in M. Points to beginning of the segment containing the innermost active function. Upon exit from a function, FBASE is restored to point to the correct base from information in the stacked function mark.
NEWIT	Iteration tag. Set to 1 at the beginning of a new nest of iterations, and used by the index unit to keep indexing straight. NEWIT is stacked in LS and restored from there each time a new iteration nest is activated.
ISMK	IS index of the marked entry closest to the top of the iteration stack. Used by IU.

B. Encodings

The APL machine makes use of a few specific encoding functions. These are used for encodings which could be expected to fit within a single machine word. Although this bias is built into the design, it is inessential to the basic ideas used in the design, and could be changed if necessary.

1. SCODE org, len, m . This is the encoding of a segment descriptor. m is 0 or 1 depending on whether this segment is for the D-machine or the E-machine. org is the beginning address and len is the length of the segment. The inverse (decoding) functions are SORG, SLEN, and SMODE, respectively. In the EM, if a segment descriptor is in QS, org is relative to its QS-index.

2. JCODE len, org, s . This is the encoding for a J-vector descriptor. The inverse functions are JLEN, JORG, JS.

3. XCODE a, b, c . Encoding used for various purposes in the E-machine. Generally, a and b are an index and its limit, respectively. c is always a single bit quantity. It is conceivable that the functions SCODE, JCODE, and XCODE might be identical in a particular implementation of the APL machine, as might their inverses. The inverse functions for XCODE are X1, X2, and X3, respectively.

4. QCODE a, b . This encoding is used in constructing ICB's during EM executions. Each field is potentially as large as the machine's memory and might be signed. The decoding functions are Q1 and Q2.

5. MCODE mask . This is the encoding function which takes a logical vector which is an access mask for an array and encodes it for storage in the AUX field of QS. The inverse function is MX1.

6. FCODE freg, iorg, name . This is the encoding used in function marks on VS. The inverses are F1, F2, F3.

C. Tags

This section summarizes the tags which can be used in VS and NT entries.

Tag	VS	NT	Meaning
UT	1	1	Undefined value.
ST	1	1	Scalar value.
JT	1	1	J-vector. Such entries are moved to QS from VS almost immediately.
DT	1	1	Descriptor array pointer. In VS means this is a result to be assigned to, while in NT, all array values have this tag. As with JT, DT entries will be deferred to QS as soon as they are noticed.
FDT	1	0	Similar to DT, except the array is to be fetched. Same note applies.
FT	0	1	Function descriptor pointer.
SGT	1	0	Segment descriptor.
NPT	1	0	Name pointer. This is an NT index.
FMT	1	0	Function mark.
RT	1	0	Unused (so far) reduction accumulator.
AT	1	0	Encoded M-address.

APPENDIX B

A FUNCTIONAL DESCRIPTION OF THE E-MACHINE

The functional description of the E-machine which follows is written in an informal dialect of APL. It differs from "standard" APL only in its sequence-controlling statements. Instead of using branches, more sophisticated, and more easily understood, constructions are utilized. These are summarized briefly below:

1. BEGIN . . . END delimits a compound statement, as in ALGOL.

2. Likewise, conditional statements and expressions of the form

   IF condition THEN . . . ELSE . . .

   are as in ALGOL. However, in this description, the condition part evaluates to 1 or 0, corresponding to TRUE or FALSE in ALGOL.

3. The case construction,

```
CASE n OF
BEGIN
   S1
   S2
   ...
   Sk
END
```

   chooses and executes the nth statement in the sequence. This description has omitted some BEGIN's and END's in compound statements within the CASE statement and substituted typographical grouping. Although this is not syntactically rigorous, it renders the description more readable.

4. The REPEAT statement repeats its range indefinitely. Within a repeated statement, the CYCLE statement is used to resume the main (compound) statement from the beginning, and LEAVE aborts the innermost REPEAT.

```
⍝      THE E-MACHINE -- A FUNCTIONAL DESCRIPTION

⍝ MAIN CYCLE ROUTINE
REPEAT
  BEGIN
  ⍝ THIS IS THE CONTROL ROUTINE IN FIGURE 2.  HOWEVER,
  ⍝ ONLY THOSE PARTS RELATED TO THE E-MACHINE ARE SHOWN.
     IF ~CASTOG THEN
        BEGIN
           IF LS[LI-1;0]≥LS[LI-1;2] THEN
              BEGIN  ⍝ TOP SEGMENT ON LS HAS OVERFLOWED
                 IF LS[LI-1;4]=1 THEN
                    BEGIN  ⍝ ITERATION MAY RECYCLE
                       LS[LI-1;0]←0
                       STEPIS
                       NEWIT ← 0
                       IF STEPTOG THEN CYCLE
                    END
                 ⍝ DEACTIVATE TOP SEGMENT AND TRY AGAIN
                 LPOP
                 CYCLE
              END
           K ← +/LS[LI-1;0,1]
           IF ~QS[K;0]∊IA,IFA,IJ,ISC,IXL THEN
              LS[LI-1;0] ← LS[LI-1;0]+1
        END
     CASTOG ← 0
     ⍝ IF ACTIVE SEGMENT IS FOR D-MACHINE THEN ACTIVATE DM
     IF LS[LI-1;3]=0 THEN DMACHINE ELSE
     CASE DECODE QS[K;0] OF  ⍝ GOES TO LABELS BELOW
     BEGIN  ⍝ DELIMITS RANGE OF  CASE  STATEMENT
     ⍝ 'LABELS' BELOW NAME E-MACHINE INTERPRETATION RULES

S)   VPUSH ST,QS[K;1]

IA ) D ← QS[K;1]
IFA) INX ← GINX K
     QS[K;2,0] ← QI, IF QS[K;0]=IA THEN A ELSE FA
     I ← S ← 0
     T ← IF LS[LI-1;7]=0 THEN NT,NLT ELSE QT,QLT
     (⍴INX) REPEAT
        BEGIN
           A ← GETDEL D,I   ⍝ A = DEL[I] FOR THIS ARRAY
           S ← S+R←IF T[0]=NT THEN A×IS[INX[I];1] ELSE 0
           QPUSH T[I=¯1+⍴INX],(QCODE R,A),INX[I],0
           I ← I+1
        END
      QS[K;1] ← QCODE (GETVBASE D),S+GETABASE D
      ERASE D

A)   IU K
FA)  VPUSH IF QS[K;0]=A THEN AT,QS[K;1]
        ELSE ST,FETCH QS[K;1]
```

```
J)    IU1 K

OP)   EXECUTE QS[K;1]  ⍝ QS[K;1] ENCODES A SCALAR OP

RED)  VPUSH RT,0
      LS[LI-1;0] ← K+QS[K;2]

DUP)  IF K>VI THEN ERROR ELSE VPUSH VS[VI-K;]

VXC)  IF VI<2 THEN ERROR ELSE VS[VI-1,2;]←VS[VI-2,1;]

POP)  VPOP

IJ)   INX ← GINX K
      S ← (JORG QS[K;1]) + IF 0=JS QS[K;1] THEN -IORG ELSE
            IORG + ¯1 + JLEN QS[K;1]
      QS[K;] ← J,(XCODE 0,S,JS D),INX,0

IXL)  QS[K;0,2] ← XL,GINX K

XL)   VPUSH ST, IF LS[LI-1;7]=0 THEN IS[QS[K;2];0] ELSE
         IORG + X1 QS[QS[K;2];]

IRP)  QS[K;] ← NIL,0,0,0

IRD)  ERASE QS[K;1]
      QS[K;] ← NIL,0,0,0

MIT)  ISMK ← II
      REPEAT
         BEGIN
            VI←VI-1
            IF VS[VI;0]=SGT THEN LEAVE
                 ELSE IF VS[VI;0]≠ST THEN ERROR
            IPUSH VS[VI;1],II=ISMK
         END
      LPUSH 0,(SORG VS[VI-1;1]),(SLEN VS[VI-1;1]),1,1,0,0
      NEWIT ← 1

SGV)  T ← QS[K;1]     ⍝ RECALL THAT SEG DESCRS ARE RELATIVE
      VPUSH SGT,SCODE (K-SORG T),(SLEN T),SMODE T

SG)   LPUSHS K

ISC)  QS[K;0,2] ← SC,GINX K

SC)   T ← IS[QS[K;2];3]∧NEWIT∨QS[K;2]≥ISMK
      IF T THEN LPUSHS K
      ELSE IF QS[K+1;0]∊XS,XC THEN
         BEGIN
            LS[LI-1;0] ← LS[LI-1;0]+1
            S ← K+1-QS[K+1;2]
            ⍝ SET CHANGE BIT TO 0
            QS[S;1] ← XCODE (X1 QS[S;1]),(X2 QS[S;1]),0
         END
```

```
JMP) IF (QS[K;0]=JMP)∨((QS[K;0]∈J0,JN0)∧VS[VI-1;1]=0)
J0 )     ∨(QS[K;0]∈J1,JN1)∧VS[VI-1;1]=1
J1 )     THEN LS[LI-1;0] ← K+QS[K;2]
JN0) IF QS[K;0]∈J0,J1 THEN VPOP
JN1)

CY)  LS[LI-1;0] ← LS[LI-1;2]

CCY) T ← K+QS[K;2]
     QS[T;1] ← XCODE(1+X1 QS[T;1]),(X2 QS[T;1]),1
     LS[LI-1;0] ← 0

RPT) LS[LI-1;0] ← 0

LVE) LPOP

CAS) IF ~(VS[VI-1;0]=ST)∧VS[VI-1;1]∈⍳QS[K;2] THEN ERROR
     LS[LI-1;0] ← K+QS[K;2]
     K ← K+VS[VI-1;1]
     VPOP
     CASTOG ← 1

XS)  J ← K-QS[K;2]
     I ← VS[VI-1;1]-IORG
     VPOP
     IF (I<0)∨I>X2 QS[J;1] THEN ERROR
        ELSE QS[J;1] ← XCODE I,(X2 QS[J;1]),1

XC)  J ← K-QS[K;2]
     QS[J;1] ← XCODE (X1 QS[J;1]),(X2 QS[J;1]),1

LX1) VPUSH ST,X1 QS[K-QS[K;2];1]

LX2) VPUSH ST,X2 QS[K-QS[K;2];1]

SX1) T ← K-QS[K;2]
     QS[T;1] ← XCODE VS[VI-1;1],(X2 QS[T;1]),1

SX2) T ← K-QS[K;2]
     QS[T;1] ← XCODE (X1 QS[T;1]),VS[VI-1;1],1

ORG) VPUSH ST,IORG

     END   ⍝ END CASE STATEMENT RANGE

END  ⍝  E-MACHINE INTERPRETATION RULES
```

```
⍝ AUXILIARY FUNCTIONS FOR E-MACHINE

∇ INX ← GINX K;R
  ⍝ INX IS A VECTOR OF QS OR IS INDICES TO ACCESS ARRAY,
  ⍝  HIGHEST COORDINATE NUMBER (I.E. FASTEST VARYING) FIRST
  R ← IF LS[LI-1;7]=0 THEN II ELSE QS[LS[LI-1;7];2]
  INX ← ⌽((Rρ2)⊤2⊥QS[K;3])/⍳R
∇

∇ LPOP
  IF LI=0 THEN ERROR ELSE LI ← LI-1
  IF LS[LI;4]=1 THEN POPIS
  IF LS[LI;5]=1 THEN FNRET
  NEWIT ← LS[LI;6]
  ⍝ IF THIS CHANGES MODES THEN CLEAN OFF QS
  IF LS[LI;3]>LS[LI-1;3] THEN
     REPEAT
        BEGIN
           IF QI = LS[LI;1] THEN LEAVE ELSE QI ← QI-1
           IF QS[QI;0] ∊ IFA,IA,RDT THEN ERASE QS[QI;1]
        END
∇

∇ POPIS
  II ← ISMK
  REPEAT
     BEGIN
        ISMK ← ISMK-1
        IF ISMK=¯1 THEN LEAVE ELSE IF IS[ISMK;4]=1 THEN LEAVE
     END
∇

∇ LPUSH V
  IF LI=LIMAX THEN ERROR
  LS[LI;⍳7] ← (6↑V),NEWIT,IF 0≠¯1↑V THEN ¯1↑V ELSE LS[LI-1;7]
  LI ← LI+1
∇

∇ LPUSHS K
  IF 0=SMODE QS[K;1] THEN ERROR
  LPUSH 0,(K-SORG QS[K;1]),(SLEN QS[K;1]),1,0,0,CORR K
∇
```

```
∇ IU1 K;T;S;R
  ⍝ CALCULATE J-VECTOR ELEMENT IN FORM  XCODE(CURR,INCR,SN)
  T ← LS[LI-1;7]
  S ← (X1 QS[K;1]),0
  IF T=0 THEN  ⍝ IF THERE IS A CHANGE, USE NEW ITER VALUE
      BEGIN
          IF IS[QS[K;2];3]∧NEWIT∨QS[K;2]≥ISMK THEN
              S ← IS[QS[K;2];0],1
      END
  ELSE IF 1=X3 QS[T+QS[K;2];1] THEN S ← (X1 QS[T+K;1]),1
  IF S[1]=1 THEN
      BEGIN
          T ← X3 QS[K;1]
          S[0] ← IF T=0 THEN S[0] ELSE -S[0]
          QS[K;1] ← XCODE S[0],(X2 QS[K;1]),T
      END
  VPUSH ST,S[0]+X2 QS[K;1]
∇

∇ IU K;IP;IQ;S;T;D
  ⍝ INDEX UNIT
  S ← 0
  IQ ← K+QS[K;2]  ⍝ BEGINNING OF ICB FOR THIS ARRAY
  T ← LS[LI-1;7]
  REPEAT
      BEGIN
          IP ← QS[IQ;2]+T
          IF T=0 THEN
              BEGIN  ⍝ THIS ARRAY INDEXED BY IS
                  IF IS[IP;3]∧NEWIT∨IP≥ISMK THEN
                      BEGIN
                          IF (IS[IP;0]=0)∧IS[IP;2]=0 THEN
                               S ← S-Q1 QS[IQ;1]
                          ELSE
                            IF (IS[IP;0]=IS[IP;1])∧IS[IP;2]=1 THEN
                               S ← S+Q1 QS[IQ;1]
                          ELSE IF IS[IP;2]=0 THEN
                               S ← S+Q2 QS[IQ;1]
                          ELSE S ← S-Q2 QS[IQ;1]
                      END
              END
          ELSE
              BEGIN  ⍝ THIS ARRAY INDEXED FROM QS
                  IF 0=X3 QS[IP;1] THEN LEAVE ELSE
                      BEGIN
                          D ← (Q2 QS[IQ;1])×X1 QS[IP;1]
                          S ← S+D-Q1 QS[IQ;1]
                          QS[IQ;1] ← QCODE D,Q2 QS[IQ;1]
                      END
              END
              IF QS[IQ;0]∊ILT,QLT THEN LEAVE ELSE IQ←IQ+1
      END
  QS[K;1] ← QCODE (Q1 QS[K;1]),S+Q2 QS[K;1]
∇
```

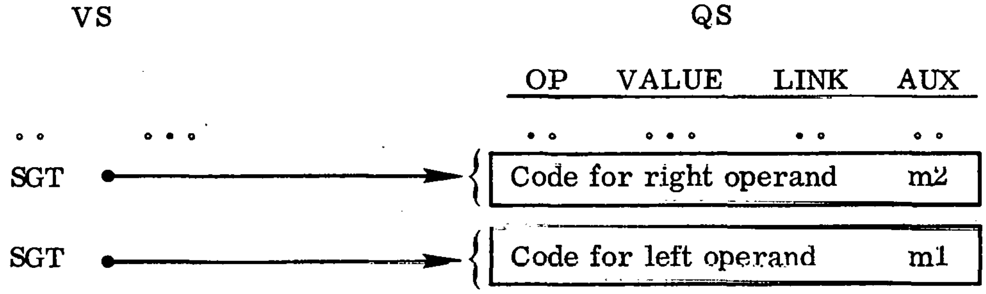
```
∇ R ← FETCH X
  ⍝ X IS A Q-CODED ADDRESS OF FORM  QCODE(VBASE,INCR)
  R ← M[1+(Q1 X)+Q2 X;]
∇

∇ EXECUTE CODOP
  ⍝ CODOP IS A DYADIC OR MONADIC SCALAR OPERATOR(ENCODED)
  ⍝ EXECUTE DECODES CODOP ON THE ELEMENTS OF VS:
  ⍝
  ⍝ IF ISDYADIC CODOP THEN
  ⍝    BEGIN
  ⍝       VS[VI-1;1] ← VS[VI-1;1] (DECODE CODOP) VS[VI-2;1]
  ⍝       VPOP
  ⍝    END
  ⍝ ELSE
  ⍝    VS[VI-1;1] ← (DECODE CODOP) VS[VI-1;1]
∇

∇ STEPIS ; I;INCR
  ⍝ STEP THE ITERATION NEST IN  IS
  ⍝ SET STEPTOG ← IF DONE THEN 0 ELSE 1
  STEPTOG ← 0
  I ← II
  REPEAT
     BEGIN
        I ← I-1
        IF (IS[I;0]=0)∧IS[I;2]=1 THEN
           BEGIN
              IF IS[I;4] THEN LEAVE ELSE
                  IS[I;0,3] ← IS[I;1],1
           END
        ELSE IF (IS[I;0]=IS[I;1])∧IS[I;2]=0 THEN
           BEGIN
              IF IS[I;4] THEN LEAVE ELSE IS[I;0,3] ← 0,1
           END
        ELSE
           BEGIN
              STEPTOG ← 1
              IS[I;3,0] ← 1,IS[I;0]
                            + IF IS[I;2]=0 THEN 1 ELSE ¯1
              LEAVE
           END
     END

∇ R ← CORR K
  R ← IF QS[K;2]=0 THEN 0 ELSE K - QS[K;2]
∇

∇ IPUSH V;MX
  ⍝ V[0] IS COUNT (SIGNED); V[1] IS MARK
     ⍝ CASE OF COUNT=0 CANNOT OCCUR (HANDLED BY D-MACHINE)
  MX ← ¯1+|V[0]
  IF II=IIMAX THEN ERROR
  IS[II;] ← (IF V[0]<0 THEN MX ELSE 0),MX,(V[0]<0),1,V[2]
  II ← II+1
∇
```

APPENDIX C

EXPANSION OF D-MACHINE OPERATORS FOR E-MACHINE

This appendix shows how the D-machine expands complex primitives into deferred sequences of E-machine instructions. It is assumed that the constraints noted for each operator are met, and that all operands have been tested for domain, conformability, and so forth before being submitted for expansion. This is not an important constraint since, for example, the requirement that an operand be beatable can always be satisfied by explicitly evaluating an unbeatable operand to temporary space.

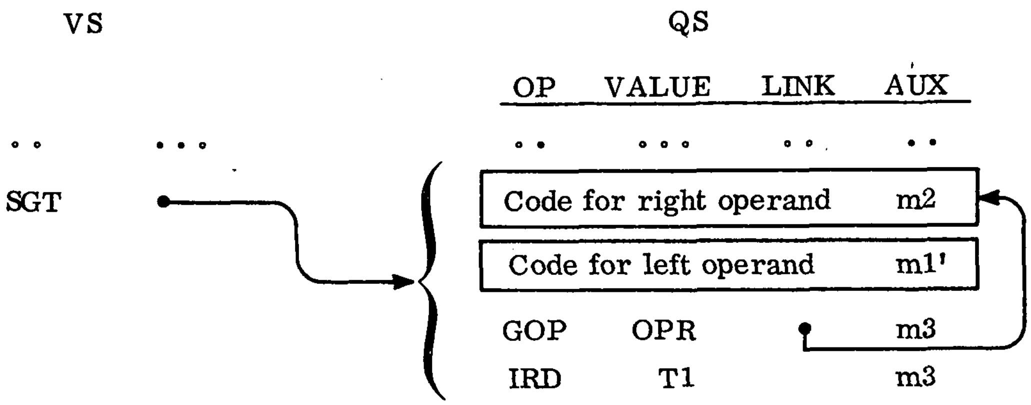
Before the expansion of any of the dyadic operations, the value stack and the instruction buffer are as follows:



where m1 and m2 are the access masks for the deferred expressions, found in the AUX field of QS. In the sequel, segments in QS are delimited graphically by braces and pointer or Greek letters are used to avoid confusion with explicit relative addressing.

1. GDF

The operands deferred in QS must be simple array values. The operand of a GDF instruction is a dyadic scalar operator, OPR. Expansion produces the following:

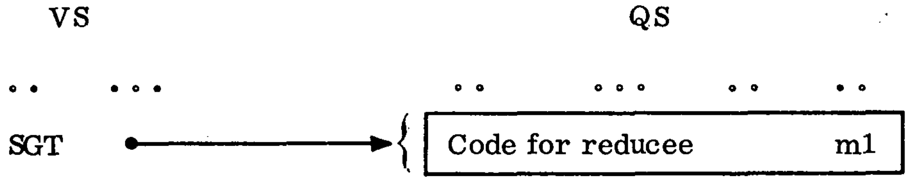


```
        VS                              QS
                             OP   VALUE   LINK   AUX

  ..        ...              ..    ...     ..     ..
  SGT                       Code for right operand    m2
                            Code for left operand     m1'
                             GOP   OPR             m3
                             IRD   T1              m3
```

In the above, T1 points to a DA containing the result rank and dimension for the GDF. m1' is m2 shifted left by the rank of the right operand. m3 is the logical <u>or</u> of m1' and m2 (i.e., m3   m1'   m2). Because of the requirement that the operands be simple array values, the segments in boxes each consist of a single IJ or IFA instruction.

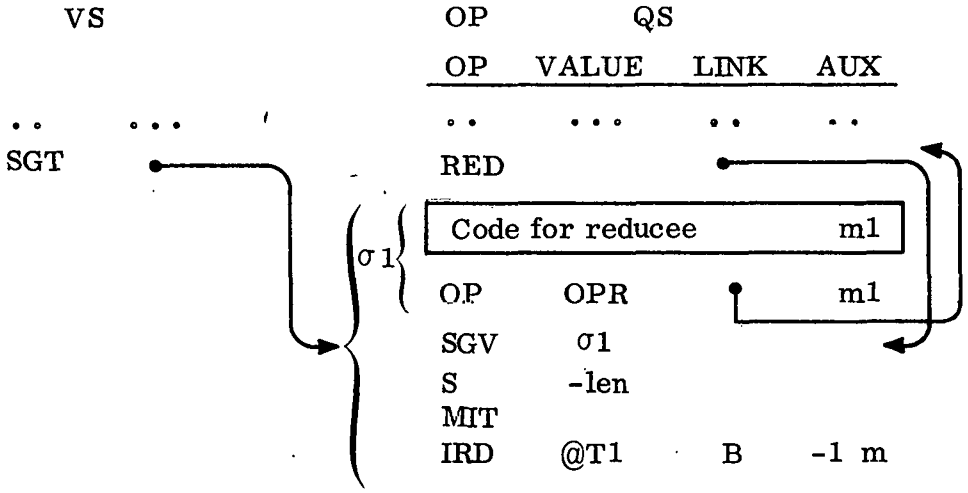
2. RED

By the time an expansion is to be done, any necessary transposes on the reducee have been performed. The variable B has value 1 if the reducee is beatable and is 0 otherwise. The "before" picture is:



```
          VS                            QS

  ..        ...              ..    ...     ..     ..
  SGT                       Code for reducee          m1
```

The reduce operator is OPR, giving rise to the expansion below:

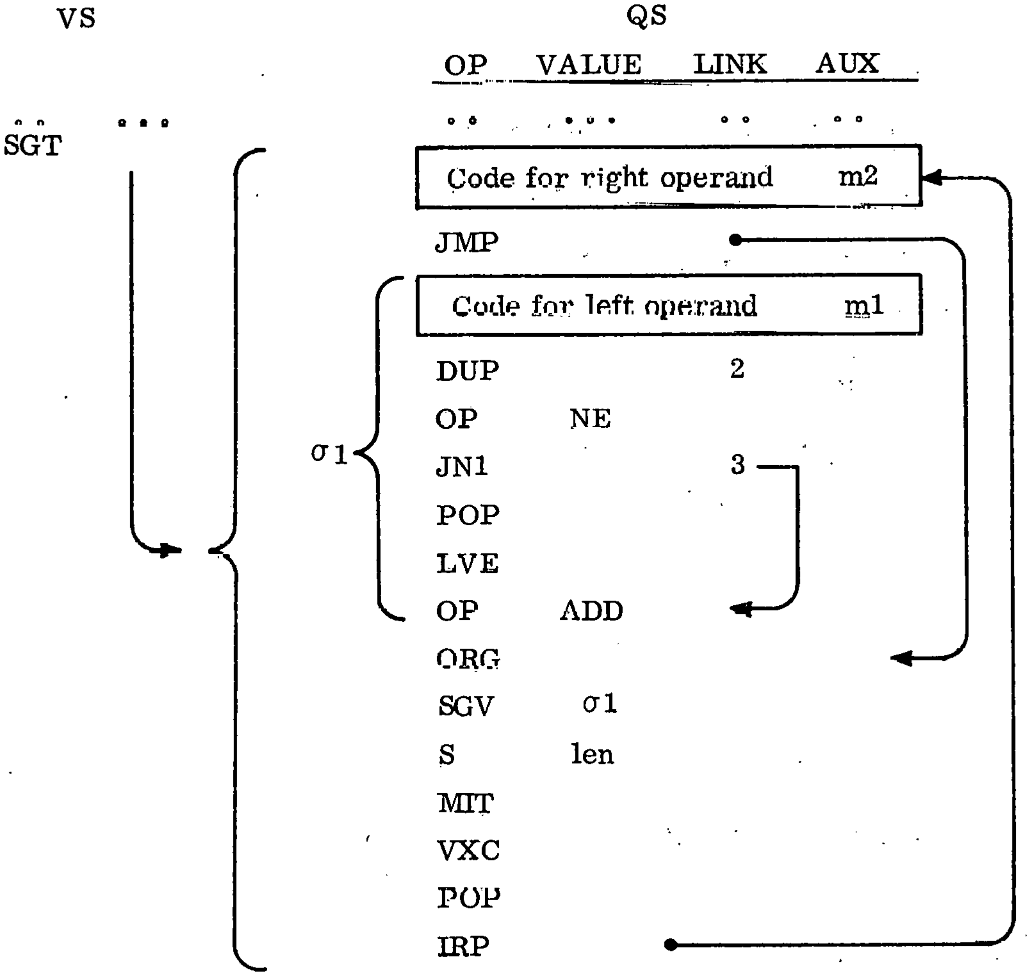


```
          VS                 OP            QS
                             OP   VALUE   LINK   AUX

  ..        ...              ..    ...     ..     ..
  SGT                        RED
                        σ1  Code for reducee          m1
                             OP    OPR               m1
                             SGV   σ1
                             S     -len
                             MIT
                             IRD   @T1      B      -1 m
```

where len is the length of the reduction coordinate and T1 is a DA with the rank and dimensions of the result.

3. DIOTA

The ranking operation, corresponding to dyadic i, requires that the left argument be a simple vector array value. This is because this operand is evaluated repeatedly during the E-machine execution of the following expansion.

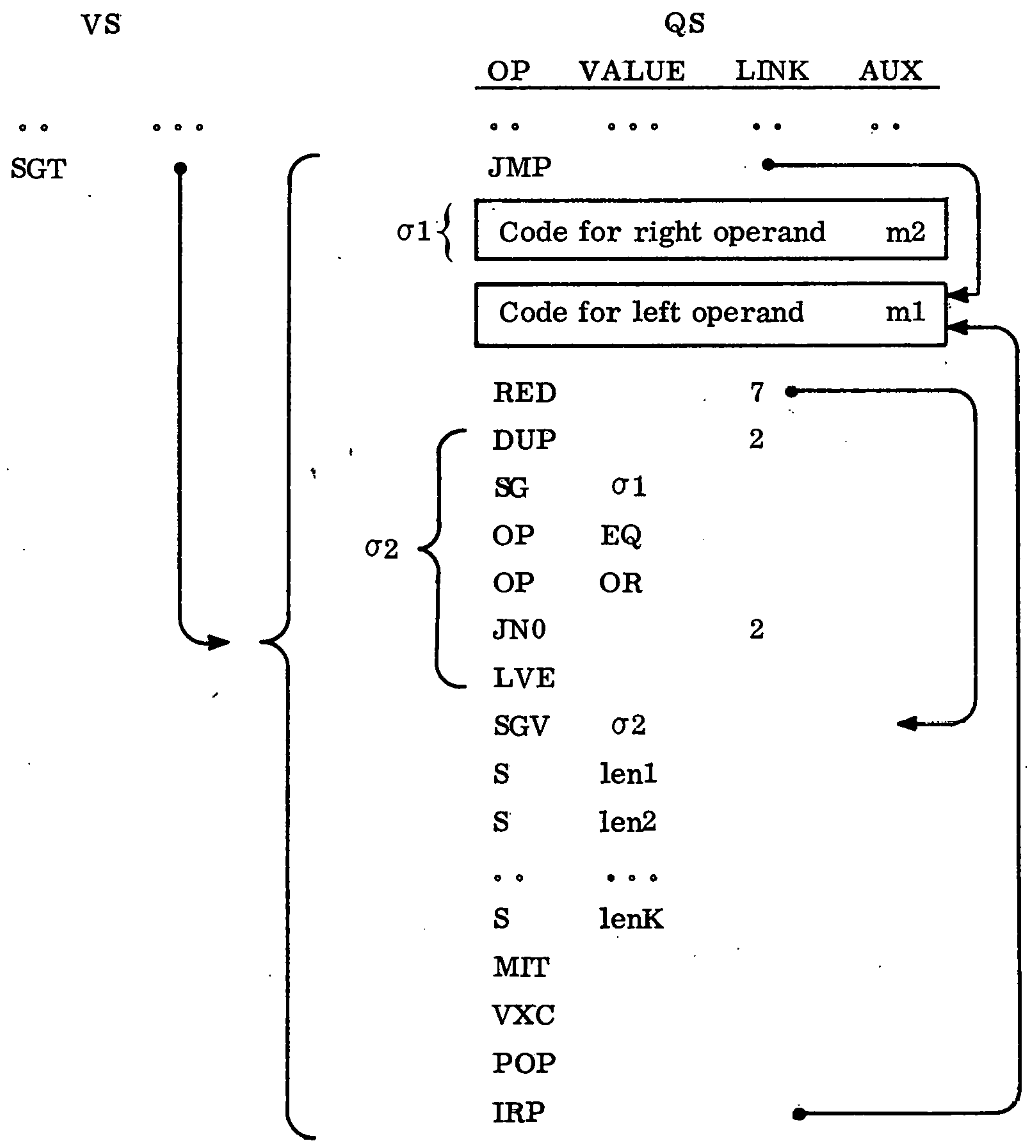


```
        VS                      QS
                         OP    VALUE   LINK   AUX
  ..      ...            ..    ...      ..     ..
  SGT
                  ┌──────────────────────────────┐
                  │ Code for right operand   m2  │
                  └──────────────────────────────┘
                         JMP
                  ┌──────────────────────────────┐
                  │ Code for left operand    m1  │
                  └──────────────────────────────┘
                         DUP            2
                         OP    NE
              σ1         JN1            3
                         POP
                         LVE
                         OP    ADD
                         ORG
                         SGV    σ1
                         S      len
                         MIT
                         VXC
                         POP
                         IRP
```

len is the length of the left operand. It should be clear from working through the above expansion that it is simply a literal interpretation in E-machine code of the definition of the ranking operator. It is assumed that the D-machine will have checked for the case of an empty vector as either operand, producing the correct result automatically. If the rank of the result is 0, that is if the right operand is a scalar, the above expansion is executed immediately by the E-machine. The IRP instruction is similar to IRD, except that it points to an instruction in QS which contains dimension information instead of referring to an explicitly-created DA.

4. EPS

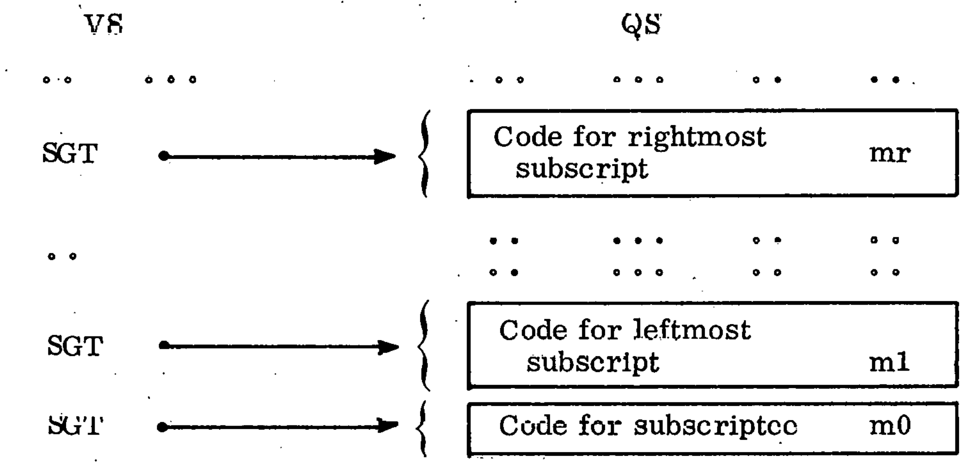
Before expanding the membership operator, a check is made for the special cases of right-operand scalar or 1-element quantity. In these cases the operation done is A=B or A=(, B)[1], respectively. Similarly, if the left operand is scalar then A=B is done. Otherwise, the expansion is made in QS as below:



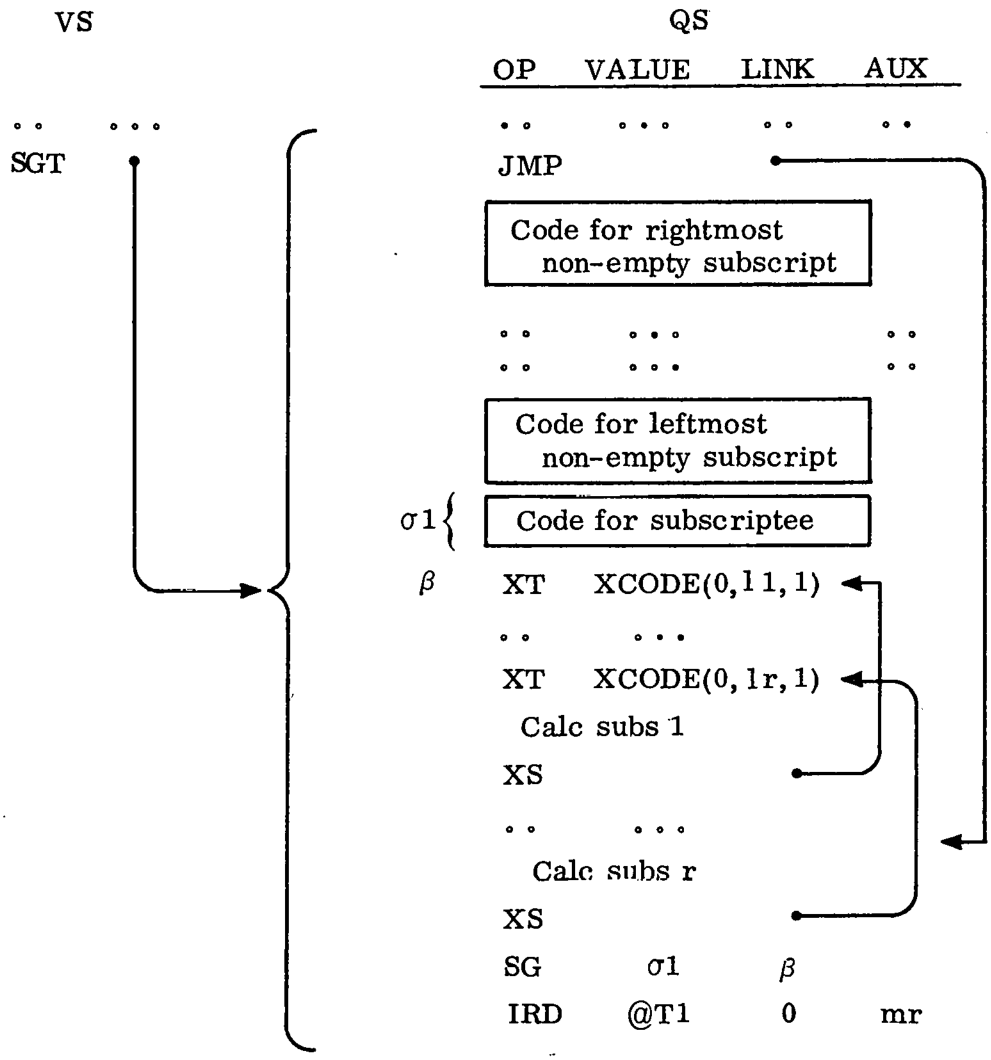
where len1, len2, . . . , lenK dimension of right operand. As in the expansion for DIOTA, the expansion of EPS is a straightforward E-machine translation of the definition of the membership operator.

5. SUBS

Before the SUBS expansion takes place, the subscripts have been examined to see if they can be beaten into the subscriptee. If an expansion is needed, then there must be some subscripts left. Before expansion, the registers contain:



The rank r of the subscriptee must be the same as the number of subscript expressions. The rank of the result is the sum of the ranks of the subscripts (counting empty subscripts as rank-1). Some of the SGT entries on the VS may be empty, that is of the form SCODE(SEG, NIL, 0). After expansion, the picture has changed to:



Where l1, l2,...,lr is the dimension of the subscriptee, minus 1. This field of the XT entries is used for checking purposes in the IU (see Section E). β is the QS index of the beginning of the XT back and @T1 is a DA with the rank and dimensions of the result. mr is the access mask of the result. The link field of β contains r, the rank of the subscriptee, which is used in the initialization of IA, IFA, IJ instructions. The lines in QS marked "Calc subs k" are one of the following:

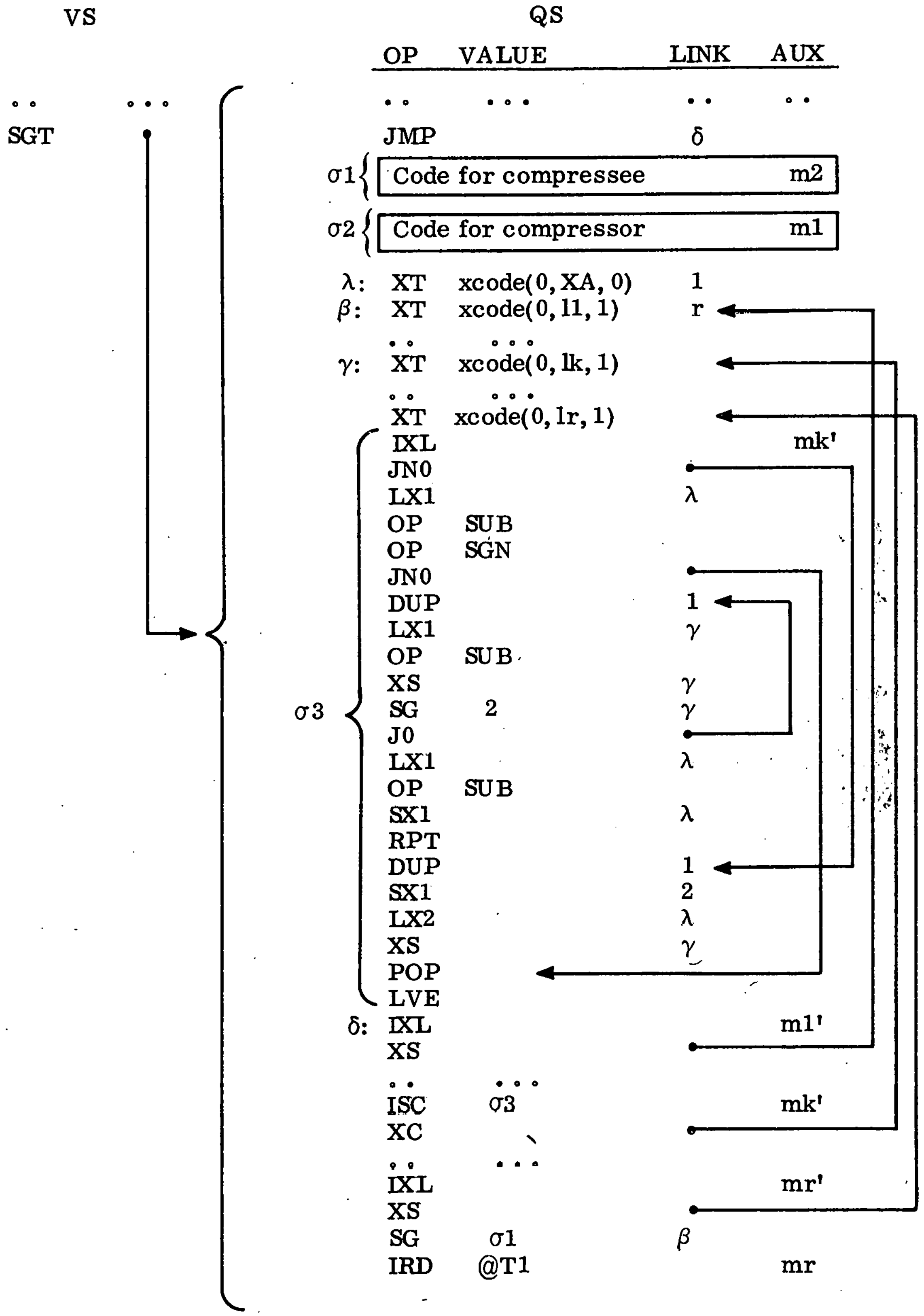
OP	VALUE	LINK	AUX
ISC	SCODE(SEG.K', 1)	0	m'
IXL	0	0	m'

In the first case, the $k^{\underline{th}}$ subscript is to be computed explicitly, which is done by activating SEG K', one of the non-empty subscript segments on QS. In the second case, the segment that was stacked on VS for this subscript was empty, so the actual subscript used is the same as that which was controlling this coordinate from the outside. The mask m' in the AUX field specifies the index environment. Example 4 in this chapter shows a specific instance of an expansion caused by the SUBS operator.

The remaining operator expansions are similar to SUBS, in that they are all special cases of it.

6. CMPRS

The compressor (left operand) has been evaluated to a temporary space, if it was not there already, and checked to see if it contains only 0 and 1 elements. In addition, the number of 1's, call it DIM1, has been counted and Vi1, the index in V of the first non-0 value is known; call it XA. This process is unfortunately necessary since we must know the rank and dimension of the result before deferral. The same process must be applied to the expansion operator. Unless the compressor falls into a special case which can be done immediately (i.e., scalar 1 or 0 or vector of all 1's or all 0's) then the following expansion is made:



```
        VS                          QS
                              OP    VALUE          LINK   AUX
                              ..    ...             ..     ..
   ..      ...
   SGT                        JMP                   δ
                       σ1 { Code for compressee            m2
                       σ2 { Code for compressor            m1
                       λ:     XT    xcode(0,XA,0)   1
                       β:     XT    xcode(0,l1,1)   r
                              ..    ...
                       γ:     XT    xcode(0,lk,1)
                              ..    ...
                              XT    xcode(0,lr,1)
                              IXL                          mk'
                              JN0
                              LX1                   λ
                              OP    SUB
                              OP    SGN
                              JN0
                              DUP                   1
                              LX1                   γ
                              OP    SUB
                              XS                    γ
                  σ3          SG    2               γ
                              J0
                              LX1                   λ
                              OP    SUB
                              SX1                   λ
                              RPT
                              DUP                   1
                              SX1                   2
                              LX2                   λ
                              XS                    γ
                              POP
                              LVE
                       δ:     IXL                          m1'
                              XS
                              ..    ...
                              ISC   σ3                     mk'
                              XC
                              ..    ...
                              IXL                          mr'
                              XS
                              SG    σ1              β
                              IRD   @T1                    mr
```

where li,...lr are as in the SUBS expansion; ml' through mr' are the masks for the individual subscripts with mk' being the mask for the compressed coordinate. The first XT entry is used to hold XA and XL where XL is the last value of the external index for the compressed coordinate. The algorithm used is as follows: Algorithm for compression: We wish to find XT such that

$$(U/[K]X)[\ldots;I;\ldots] \leftrightarrow X[\ldots;XT;\ldots]$$

Let XL be the last value of I from which the last XT was calculated. XA is the index of the first 1 in U. Then, the QS expansion for compression calculates the new value of XT as a function of the new I and old XT and XL as follows:

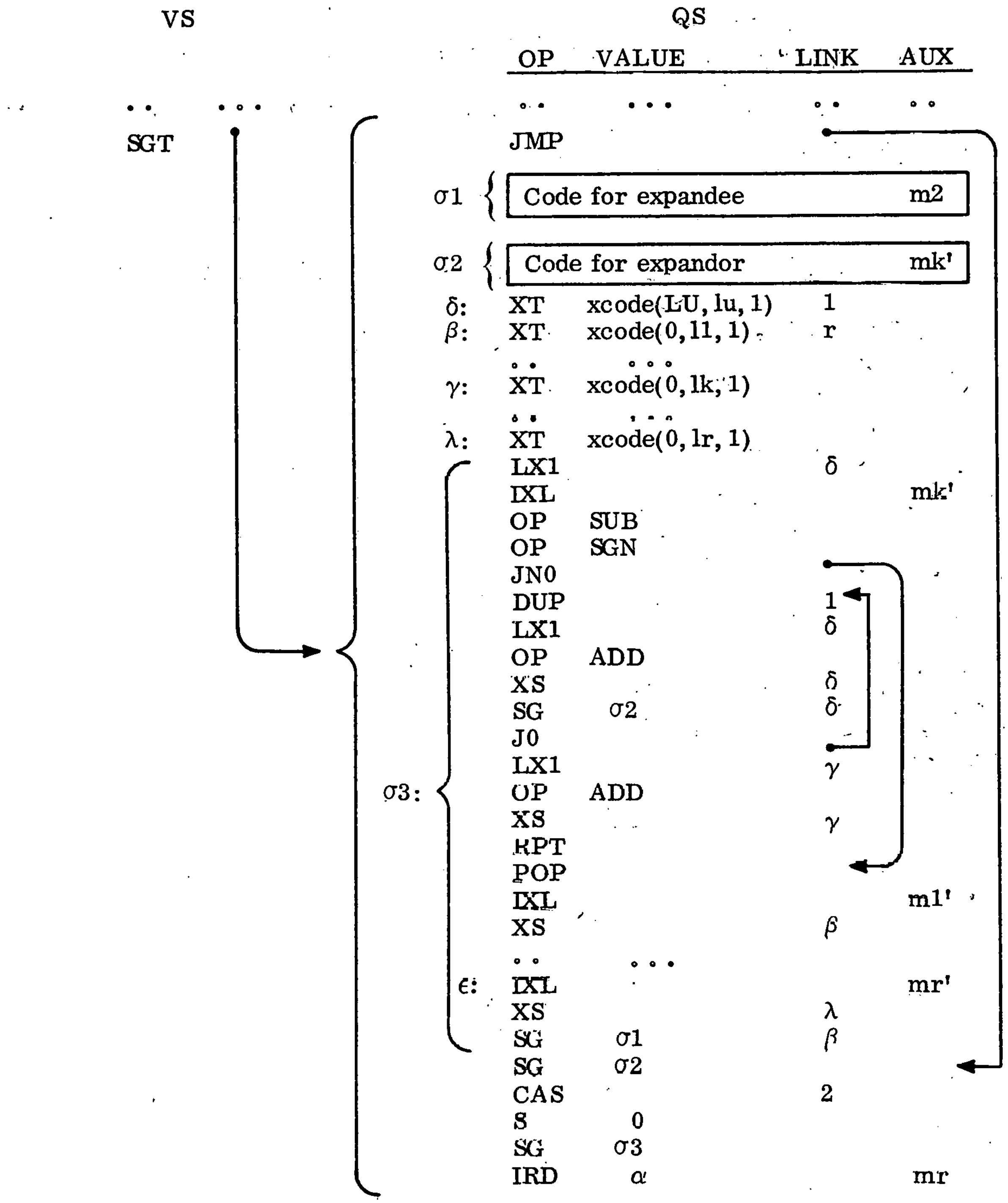
```
if I=0 then
    begin
        XL←0
        XT←XA
    end
else
    repeat
        begin
            T←×XL-I
            if T=0 then leave
            repeat
                begin
                    XT←XT-T
                    if U[XT]=1 then leave
                end
           XL←XL-T
        end
```

7. EXPND

The EXPND operator is treated similarly to CMPRS. In particular, the expandor (left operand) is checked to see that it is a logical quantity and the number of 1's is compared to the length of the expansion coordinate. If the expandor falls into one of the special cases (all ones, all zeros) the result is calculated immediately. Otherwise, the QS expansion that follows is made to implement the expansion algorithm below:

Let R be (U/[K]X)[...;I;...]. Then we want to find LX such that R←if U[I]=0 then 0 else X[...;LX;...]. LU is the index of the last found 1 in U and LX is the corresponding X index (on the Kth coordinate).

```
if U[I]=0 then R←0 else
    begin
        repeat
            begin
                T←×I-LU
                if T=0 then leave
                repeat
                    begin
                        LU←LU+T
                        if U[LU]=1 then leave
                    end
                LX←LX+T
            end comment main repeat;
            R←X[...;LX;...]
    end
```



```
      VS                                 QS
                                   OP    VALUE              LINK   AUX
 . .     . . .                     . .   . . .              . .    . .
 SGT                               JMP
                          σ1       Code for expandee                m2
                          σ2       Code for expandor                mk'
                          δ:       XT    xcode(LU, lu, 1)   1
                          β:       XT    xcode(0, l1, 1)    r
                                   . .   . . .
                          γ:       XT    xcode(0, lk, 1)
                                   . .   . . .
                          λ:       XT    xcode(0, lr, 1)
                                   LX1                      δ
                                   IXL                             mk'
                                   OP    SUB
                                   OP    SGN
                                   JN0
                                   DUP                      1
                                   LX1                      δ
                                   OP    ADD
                                   XS                       δ
                                   SG     σ2                δ
                                   J0
                                   LX1                      γ
                     σ3:           OP    ADD
                                   XS                       γ
                                   RPT
                                   POP
                                   IXL                             ml'
                                   XS                       β
                                   . .   . . .
                          ε:       IXL                             mr'
                                   XS                       λ
                                   SG     σ1                β
                                   SG     σ2
                                   CAS                      2
                                   S      0
                                   SG     σ3
                                   IRD    α                        mr
```

Note that the sequence of IXL and XS instructions starting at ε does not contain a reference to the kth subscript position as this has already been computed at the beginning of the segment activated by the CAS instruction. Also, in the above, the quantity ℓu in the X2 field of the pseudo-iteration stack at is the length of vector U_J less 1.

8. ROT

Rotation is a special case of subscripting defined as follows:

If N is a scalar, then R←N⌽[K]M means for each L ELT ⍳⍴M

R[;/L]↔M[;/((K-1)↑L),(IORG+(⍴M)[K]|(N-IORG)+⍳(⍴M)[K]),K↓L]

If N is an integer array with ⍴N↔(K≠⍳⍴⍴M)/⍴M then

R[;/L]↔M[;/((K-1)↑L),(IORG+(⍴M)[K]|(N[;/L']-IORG)+⍳(⍴M)[K]),K↓L]

where L'↔(K≠⍳⍴⍴M)/L.

Thus the expansion for ROT in QS is the same as for a general subscript with all but the $K^{\underline{th}}$ coordinate being IXL, XS pairs and the $K^{\underline{th}}$ coordinate being computed according to the above definition. The explicit expansion will be omitted since it is similar to what has already been shown.

APPENDIX D

POWERS OF 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

CHAPTER V

EVALUATION

In this chapter we examine the design for an APL machine proposed in Chapter IV and compare its performance to more conventional architectures. This is done by showing that the APLM is more efficient in its use of memory than a less sophisticated computer doing the same task.

A. Rationale

In Chapter III, a number of design goals for the APLM were stated:

1. Machine language should be "close" to APL.
2. Machine should be general, flexible.
3. Machine should do as much as possible automatically.
4. Machine should expend effort proportional to the complexity of its task.
5. Design should be elegant, clean, perspicuous.
6. Machine should be efficient. In particular, it should be parsimonious of memory allocation and accessing.

We can dispose of some of these in short order. To begin with, goals 1, 3, and 4 have obviously been satisfied. Since the machine designed implements APL, to goal 2 we can reply that the machine is general and flexible at least to the extent that APL as a language is general and flexible. For example, even though the APLM does not include all of the LISP primitives, if it is easy to write a LISP interpreter in APL, then the machine should be able to handle them with ease.

Although I believe that the goal of elegance has been satisfied, this is not the place to make such judgements, nor am I the one to make them. This particular aspect will have to be decided by less prejudiced readers. A seventh, unstated goal is that the design should indeed work. It should be clear to the reader who has reached this point that the basic machine structure proposed is in fact sound and that an APL machine as described will produce correct answers.

This leaves the question of efficiency to be considered. Because we have not detailed a complete machine, traditional measures such as encoding efficiencies of comparisons of cycle times cannot be used. A major emphasis throughout this work has been to minimize the necessity for temporary storage in expression evaluation and simultaneously to minimize memory accessing. While these problems are often of marginal importance in a conventional design, they are quite significant in an APL machine, since operands are generally arrays. Thus a temporary store is no longer a single word, but is potentially an array of indefinite size. Similarly, the conventional problem of saving a single fetch where a quantity might be in a register, becomes the problem of saving 1000 fetches for an array operand.

The remainder of this chapter is dedicated to the evaluation of machine efficiency. We take an analytic approach here, but cannot hope to have a simple analytic model of the machine per se which would give clean, closed-form quantitative data about the APLM. Instead, the analysis compares the performance of the APLM to a fictitious "naive machine," which is simply a straightforward interpreter of the semantics of APL.

The next section discusses the naive machine (NM) and outlines the assumptions upon which the comparisons will be based. In the sequel, we will compare the two machines by looking at the number of individual fetches, stores, operations, and temporary stores needed to do a particular task. Different tasks will be examined with this in mind. At the end of the chapter, these results will be summarized together with some conclusions.

B. The Naive Machine

Although the APL machine proposed in Chapter IV has never been implemented, there exist concrete examples of the naive machine. These include APL\7090

(Abrams [1966]), APL\1130 (Berry [1968]), and APL\360 (Falkoff and Iverson [1968]; Pakin [1968]). The main feature which distinguishes the NM from the APLM is that the APLM defers many computations while the naive machine evaluates each subexpression immediately after its operands have been evaluated. The APLM, by contrast, does some of its evaluations immediately (e.g., scalar results), defers some indefinitely (by drag-along), and does still others in a non-direct way (e.g., beating).

The following list of assumptions clarifies in more detail the differences between the APLM designed in this work and our "standard" naive machine as used in the rest of this chapter.

1. The naive machine uses the same representation for arrays as does the APL machine. If the naive machine is APL\360, then this is approximately true. In fact, APL\360 does not separate DA's from value parts in array representations. On the other hand, APL\360 represents scalars as rank-0 arrays, and is thus more inefficient in its handling of scalar values. We assume here that the NM keeps scalar values in a value stack as does the APLM. We have also (generously) assumed that the NM uses the J-vector representation for interval vectors. In general, these assumptions cast the naive machine in a better light than any current implementation of APL.

2. The naive machine generates a result value whenever an operator is found and its operands are evaluated. (This is exactly the way APL\360 works.) Further, we assume that the NM will use temporary space allocated to one of its operands for the result, if possible; e.g., if the expression A+B is to be evaluated, a new temporary space must be found to accommodate the result. However, if the expression is A+B+C; the subexpression B+C will be evaluated first causing the creation of a temporary t which can then be used as the result destination for the value of A+t.

3. In an assignment to a variable, as in A←expression, the naive machine performs the assignment simply by storing a pointer to the temporary for the evaluated expression in the nametable entry for A. Again, this is consistent with the functioning of APL\360.

4. Each operation in either the NM or the APLM requires a fixed amount of overhead (e.g., rank checking, domain checking, space allocation, setup, drag-along, etc.). An analysis of the instructions for both machines shows that these processes take approximately the same effort in both machines. Since there is no way to compare this effort with the memory usage measures discussed here, it will be omitted. For a single statement, this overhead appears as a linear additive term.

5. Since scalars are kept in the value stack in both machines and since the VS mechanism is not specified (e.g., it could be a hard-wired stack, or a fast scratchpad memory, or it could be kept in memory with other array values), all scalar fetches and stores will be ignored. The effort to evaluate array expressions always dominates the effort for scalar expressions.

6. There are no distinctions made between data types in the APL machine. We thus assume that both the APLM and the NM use the same representation for individual data elements.

7. All scalar operations take the same amount of time to perform. That is, an add or a multiply will each be counted as a single operation.

8. Finally, it is assumed that both the naive machine and the APL machine are implemented in similar technologies so that the cost of memory accesses, storage allocations, and operations are the same for both machines.

C. Analysis of Drag-Along And Beating

To begin the analysis, let us look at a subset of the operations of APL and derive some analytic results comparing the APLM and the NM. The set to be considered is

1. Selection operations
2. Monadic and dyadic scalar arithmetic operations
3. Inner products
4. Reductions of the above (this includes outer products)
5. Assignments of above to unconditioned variables or to variables conditioned by selection operators.

We consider only those expressions which are array-valued, as scalar expressions are done similarly in both machines. Each operation requires the machine evaluating it to do a certain amount of work, summarized in Table 1 below. Tables 2A and 2B summarize the "effort" required to do these manipulations.

In Table 2, some of the entries contain conditional terms or factors. These account for the different possible initial conditions when a subexpression is evaluated. Also, notice that in Table 2B, some of the entries contain references to the functions DOF, DOS, and DOO. These are functions which, given a deferred expression as argument, return as values the number of fetches, stores, and operations, respectively, necessary to evaluate the expression. Thus, for the APL machine, Table 2B does not tell the whole story; we must also take into account the efforts to evaluate the final deferred expression (by the E-machine). Hence, it is necessary to give detailed definitions of the DOF, DOS, and DOO functions.

TABLE 1

Steps in Evaluation of APL Operators

NAIVE MACHINE	APL MACHINE
A. Selection Operators	
1. Check rank, domain of operands.	1. Check rank, domain of operands.
2. Get space for result DA, value.	2. Get space for result DA (if operand is a variable).
3. Set up DA, M-headers.	3. Set up DA.
4. Set up copy operation.	4. Adjust VS, QS.
5. Do copy operation.	
6. Adjust VS.	
B. Monadic Scalar Operators	
1. Get space for result DA, value (only if operand is a variable).	1. Defer operation to QS.
2. Set up DA, M-headers if space was gotten in step 1.	2. Adjust VS, QS.
3. Do the operation.	
4. Adjust VS.	
C. Dyadic Scalar Operators	
1. Check rank, dimensions of operands.	1. Check rank, dimensions of operands.
2. Get space for result DA, value (only if both operands are variables).	2. If one operand is a scalar, move it to QS.
3. Set up DA, M-headers if space was gotten in step 2.	3. Defer operation to QS.
4. Do the operation.	4. Adjust VS, QS.
5. Adjust VS.	
D. Outer Product	
1. Get space for result DA, value.	1. If operands are deferred subexpressions, then evaluate them to temp space.
2. Set up DA, M-headers.	2. Get space for result DA.
3. Do the operation.	3. Set up DA.
4. Adjust VS.	4. Defer operation to QS.
	5. Adjust VS, QS.

Table 1 (cont.).

NAIVE MACHINE	APL MACHINE
E. Reduction	
1. Get space for result DA, value.	1. Get space for result DA.
2. Set up DA, M-headers.	2. If reduction coordinate is other than the last, then do appropriate transpose.
3. Do the reduction.	3. Set up DA.
4. Adjust VS.	4. Defer operation to QS.
	5. Adjust VS, QS.
F. Assignment to Simple Variable	
1. If right-hand side is a temp then go to step 6, otherwise do steps 2 through 7.	1. If right-hand side is a temp then go to step 6, else proceed.
2. Get space for DA, value.	2. If the LHS* variable is already defined and is of the correct size and does not appear permuted as an operand in the deferred RHS then go to step 5.
3. Set up DA, M-headers.	3. Get space for DA, value of LHS.
4. Set up copy operation.	4. Set up DA and M-headers.
5. Do copy operation.	5. Defer operation in QS.
6. Adjust VS.	6. Adjust VS, QS.
7. Adjust Nametable.	7. Adjust Nametable.
G. Assignment to a Selected Variable	
1. Check dimensions of LHS, RHS.	1. Check dimensions of LHS, RHS.
2. Set up copy operation.	2. If RHS contains deferred instances of LHS variable which are permuted differently than LHS, then proceed else go to step 6.
3. Do copy operation.	3. Get space for DA, value of RHS.
4. Adjust VS.	4. Set up DA, M-headers.
	5. Evaluate RHS to this temp.
	6. Defer selected assignment to QS.
	7. Adjust VS, QS.

*LHS and RHS refer to the left-hand side and right-hand side of an assignment arrow, respectively.

TABLE 2A

Summary of Effort to Evaluate Operators - NAIVE MACHINE

OPERATOR	FETCHES	STORES	TEMPS	OPERATIONS
SELECTION (*R IS*: sel ℰ)	×/ρ*R*	4+(ρρ*R*)+×/ρ*R*	*P*1×(4+(ρρ*R*)+×/ρ*R*)	0
SCALAR MONADIC (*R IS*: *OP* ℰ)	×/ρ*R*	(*P*1×(4+ρρ*R*))+×/ρ*R*	*P*1×(4-(ρρ*R*)+×/ρ*R*)	×/ρ*R*
SCALAR DYADIC (*R IS*: ℰ *OP* ℱ)	*N*1××/ρ*R*	(*P*2×(4+ρρ*R*))+×/ρ*R*	*P*2×(4-(ρρ*R*)+×/ρ*R*)	×/ρ*R*
OUTER PRODUCT (*R IS*: ℰ ∘.*OP* ℱ)	(×/ρ ℰ)+×/ρ*R*	4+(ρρ*R*)÷×/ρ*R*	4+(ρρ*R*)+×/ρ*R*	×/ρ*R*
REDUCTION (*R IS*: *OP*/[*K*] ℰ)	×/ρ ℰ	4+(ρρ*R*)+×/ρ*R*	4+(ρρ*R*)+×/ρ*R*	×/ρℰ
ASSIGNMENT *A*←ℰ	*P*1××/ρℰ	*P*1×(4+(ρρ ℰ)+×/ρ ℰ)	*P*1×(4+(ρρ ℰ)+×/ρ ℰ)	0
ASSIGNMENT (sel *A*)←ℰ	×/ρ sel *A*	×/ρ sel *A*	0	0

Notes: P1 ⟷ if ℰ is a variable then 1 else 0. P2 ⟷ if ℰ and ℱ are both variables then 1 else 0 .

N1 ⟷ if ℰ and ℱ are both arrays then 2 else 1.

TABLE 2B

Summary of Effort to Evaluate Operators - APL MACHINE

OPERATOR	FETCHES	STORES	TEMPS	OPERATIONS
SELECTION (*R IS*: sel $\mathscr{E}$)	0	*N*1×(3+ρρ*R*)	*N*2×(3+ρρ*R*)	0
SCALAR MONADIC (*R IS*: *OP* $\mathscr{E}$)	0	0	0	0
SCALAR DYADIC (*R IS*: $\mathscr{E}$ *OP* $\mathscr{F}$)	0	0	0	0
OUTER PRODUCT (*R IS*: $\mathscr{E}$∘.*OP*$\mathscr{F}$)	(*P*1×*DOF*($\mathscr{E}$))+(*P*2×*DOF*($\mathscr{F}$))	3+(ρρ*R*)+(*P*1×*DOS*($\mathscr{E}$)) +(*P*2×*DOS*($\mathscr{F}$))	3+ρρ*R*	(*P*1×*DOO*($\mathscr{E}$)) +(*P*2×*DOO*($\mathscr{F}$))
REDUCTION (*R IS*: *OP*/[*K*]$\mathscr{E}$)	0	3+(ρρ*R*)+*P*3×*N*1×(4+ρρ*R*)	3+(ρρ*R*)+*P*3×*N*1×(3+ρρ*R*)	0
ASSIGNMENT *A*←$\mathscr{E}$	0	*P*4×(4+ ρρ$\mathscr{E}$)	*P*4×(4+(ρρ$\mathscr{E}$)+×/ρ$\mathscr{E}$)	0
ASSIGNMENT (sel *A*)←$\mathscr{E}$	*P*5×*DOF*($\mathscr{E}$)	*P*5×(*DOS*($\mathscr{E}$)+4+(ρρ$\mathscr{E}$)+×/ρ$\mathscr{E}$)	*P*5×(4+(ρρ$\mathscr{E}$)+×/ρ$\mathscr{E}$)	*P*5×*DOO*($\mathscr{E}$)

NOTES: N1 ⟷ Number of array opnds in $\mathscr{E}$
P1 ⟷ if $\mathscr{E}$ contains deferred operators then 1 else 0
P3 ⟷ if K≠⌈/ιρρ$\mathscr{E}$ then 1 else 0
P5 ⟷ if $\mathscr{E}$ must be evaluated first then 1 else 0

N2 ⟷ Number of opnds with reference count > 1
P2 ⟷ if $\mathscr{F}$ contains deferred operators then 1 else 0
P4 ⟷ if $\mathscr{E}$ is a temp or A is defined and of correct size and there are no indexing conflicts then 0 else 1

For the set of expressions containing only selection operations, scalar arithmetic operations, outer products, reductions, and assignment, it is relatively simple to specify the DOF, DOS, and DOO functions. Recall that in the APL machine, expressions are deferred in QS, which contains an operation code and an access mask for each entry. Let the function OP(I) be the operation code for QS[I;] and MASK(I) have as its value the access mask in the AUX field of QS[I;]. Finally, for a given expression in QS, let RR be the dimension of the final result. For each QS entry whose opcode is IFA, IA, OP, or GOP define the function D(I) whose value is a dimension vector as follows: if the entry is not within a reduce segment then D(I) is RR. Otherwise catenate an element with the length of each reduction coordinate; the innermost reduction corresponds to the last element of D(I). Thus, D(I) is the vector of limits of the iteration stack which are active when instruction QS[I;] is executed by the E-machine. The idea here is that D(I) represents the indexing environment of QS[I;]. If N(I) is the index of the rightmost 1 in MASK(I) (that is, N(I)←→⌈/(MASK(I))/ιρMASK(I)), then the following algorithm calculates the desired functions:

```
RF←RS←RO←0
I←starting addr of deferred expression in QS
repeat
    begin
        if OP(I) = IFA then RF← RF+×/N(I)↑D(I)
    else if OP(I)=IA then RS←RS+×/N(I)↑D(I)
    else if OP(I)∈OP, GOP then RO←RS+×/N(I)↑D(I)
        I←I+1
        if I > segment ending addr then leave
    end
```

Then DOF(ℰ)←→RF; DOS(ℰ)←→RS; DOO(ℰ)←→RO.

D. Example — A Simple Subclass of Expressions

Since the input to either the naive machine or the APL Machine may be any arbitrary expression,it is difficult to produce a closed-form comparison of the performance of the two. However, we can look in detail at a simple subset of expressions and obtain some estimates on how the two machines compare. Consider the set of expressions of the form $A \leftarrow \mathcal{E}$, where $\mathcal{E}$ is an expression containing only array-shaped operands combined by scalar arithmetic operators and selection operators. As an aid to the analysis, construct the tree corresponding to the expression $\mathcal{E}$, and number all the nodes corresponding to operators. Then, construct vectors RR, RD, TY, TV, $N1$ and $N2$ as follows:

For each node I, representing $RESULT\ T \leftarrow \mathcal{E}'$, where $\mathcal{E}'$ is the subexpression rooted at node I,

$RD[I] \leftarrow \times/\rho RESULT$ (Result Dimension of node I)

$RR[I] \leftarrow \rho\rho RESULT$ (Result Rank of node I)

$TY[I] \leftarrow$ if operator is a select then $^{-}1$ else if monadic then 1 else 2

$TV[I] \leftarrow$ if all sons of node I are variable names then 1 else 0

$N1[I] \leftarrow$ number of leaves in the subtree of node I

$N2[I] \leftarrow$ number of leaves in the subtree of node I accessible through a path not including a select operation.

Finally, let R be the number of array operands in $\mathcal{E}$

M be the number of monadic scalar operators in $\mathcal{E}$ (i.e., $+/1=TY$)

N be the number of dyadic scalar operators in $\mathcal{E}$ (i.e., $+/2=TY$)

S be the number of selection operators in $\mathcal{E}$ (i.e., $+/^{-}1=TY$)

Z be the number of elements in $\mathcal{E}$ (i.e., $\times/\rho\mathcal{E}$)

Y be the rank of $\mathcal{E}$ (i.e., $\rho\rho\mathcal{E}$)

P be: if APLM must get space for A then 1 else 0.

Note that in a wcll-formod expreesion $N=R-1$.

Then, from Tables 2A and 2B, and the definitions of DOO, DOS, and DOF, we see that the effort for each machine to evaluate ℰ is as follows:

NAIVE MACHINE

fetches: +/*RD*×|*TY*

stores: (+/*RD*)++/((¯1=*TY*)∨*TV*∧(1≤*TY*))/(4+*RR*)

temps: +/*TV*/(4+*RR*+*RD*)

operations: +/(1≤*TY*)/*RD*

APL MACHINE

fetches: *R*×*Z*

stores: *Z*+(*P*×(4+*Y*))++/(¯1=*TY*)/*N*1×(3+*RR*)

temps: (*P*×(4+*Y*+*Z*))++/(¯1=*TY*)/*N*2×(3+*RR*)

operations: +/(1≤*TY*)/*Z*

In general, each formula above is the sum of the relevant entries in Tables 2A or 2B. As the fetch formulas are obvious, we show the derivation of the store count for the NM. First, each operator in ℰ calculates a result which must be stored immediately which gives the term +/*RD*. Also, temporary space must be allocated for selection operations and those cases of scalar operators in which one of the operands is not itself a temporary. In such a case, another 4+ (result-rank) words must be stored. (All but one of these is for the new DA; the other is for the header word for the value array.) The result ranks of the operations in ℰ are in the vector *RR*. Thus, the compression selects those elements of 4+*RR* which correspond to the conditions just stated. In particular, (¯1=*TY*) is a vector having a one for each selection operator and *TV*∧(1≤*TY*) has a one for each monadic or dyadic scalar operator whose evaluation requires temporary space to be allocated. The sum of these terms gives the formula shown; the other formulas are derived similarly.

We can form the ratios of the corresponding quantities for each machine and attempt to get some estimate of their values. RF, the ratio of fetches in the naive machine to fetches in the APL machine,is given by:

$$RF \leftrightarrow \frac{+/RD\times|TY}{R\times Z} \geq \frac{+/Z\times TY}{R\times Z} \text{ since } Z\leq RD.$$

$$\leftrightarrow \frac{Z\times+/|TY}{R\times Z} \leftrightarrow \frac{Z\times(M+S+2\times N)}{R\times Z} \leftrightarrow \frac{M+S+(2\times R)-2}{R} \quad \text{because} \quad N=R-1$$

Thus, $RF \geq 2+\frac{M+S-2}{R}$

Hence, for fetches, the APLM does at least twice as well as the NM if there are at least two monadic or select operators. The worst case is when M or S or N is 1 and the rest are 0, in which case the ratio is 1. The above also shows that the ratio increases (without bound) in proportion to the number of monadic and select operators in the expression $\mathscr{E}$.

The ratio of stores for the two machines, RS , is:

$$RS \leftrightarrow \frac{(+/RD)++/((\bar{1}=TY)\vee TV\wedge(1\leq TY))/(4+RR)}{Z+(P\times(4+Y))++/(\bar{1}=TY)/N1\times(3+RR)}$$

$$\geq \frac{(Z\times\rho RD)++/((\bar{1}=TY)\vee TV\wedge(1\leq TY))/(4+RR)}{Z+(P\times(4+Y))++/(\bar{1}=TY)/N1\times(3+RR)}$$

$$\leftrightarrow \frac{(M+N+S)+\dfrac{+/((\bar{1}=TY)\vee TV\wedge(1\leq TY))/(4+RR)}{Z}}{1+\dfrac{(P\times(4+Y))++/(\bar{1}=TY)/N1\times(3+RR)}{Z}}$$

$(SINCE\ \rho RD \leftrightarrow M+N+S)$

But the numerators of the two fractions with denominator Z are bounded, while Z can increase without bounds. Thus for large Z,

$$RS\approx M+N+S$$

That is, in expressions in which the size of the operand arrays is large (i.e., at least as many elements as there are operators) the NM requires more stores than the APLM, approximately in proportion to the number of operators in the expression.

In the case of temporary storage allocated, the ratio, RT, is:

$$RT \leftrightarrow \frac{+/TV/(4+RD+RR)}{(P\times(4+Y+Z))++/({}^{-}1=TY)/N2\times(3+RR)}$$

$$\geq \frac{+/TV/(\rho RD)\rho(4+Y+Z)}{(4+Y+Z)++/({}^{-}1=TY)/N2\times(3+RR)}$$

$$\leftrightarrow \frac{+/TV}{1+\frac{+/({}^{-}1=TY)/N2\times(3+RR)}{4+Y+Z}}$$

Again, the lower bound is greater than 1, since $(+/TV)\geq 1$. In this case, the ratio is of the order of $+/TV$, for large Z, which is a function of the tree structure of $\mathscr{E}$ rather than an explicit function of its operator count. Note that in the case where $\mathscr{E}$ contains no select operations and P is 0, the ratio is infinite, since the APLM requires no temporary storage.

For the case of operations the ratio, RO, is:

$$RO \leftrightarrow \frac{+/(1\leq TY)/RD}{+/(1TY)/Z}$$

But $Z\leq RD$ and the compression in both numerator and denominator select the same terms. Thus, $RO\geq 1$.

E. Example — An APL One-Liner

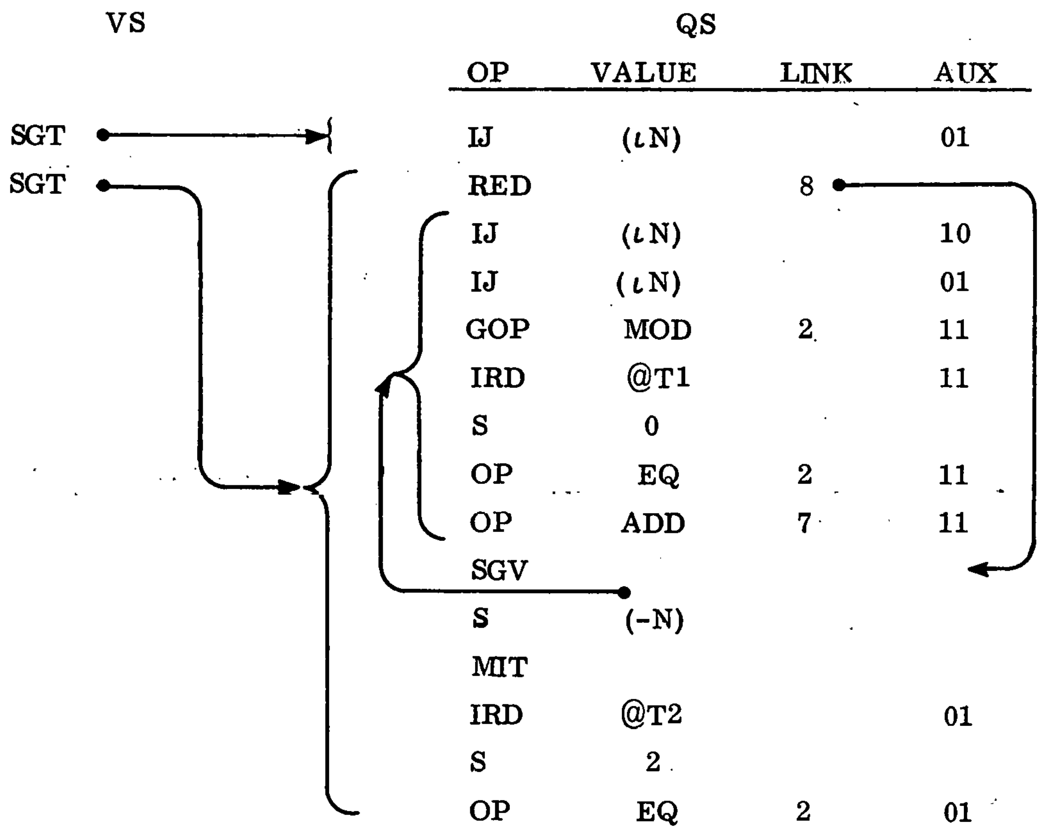
APL makes it easy to produce simple one-line programs to do some interesting task. One such is the program (expression) for finding all the prime numbers less than or equal to N, as shown below. (Index origin is 1)

```
PRIMES ← (2=+/[1]0=(⍳N)∘.|⍳N)/⍳N
```

Although the algorithm used is clearly inefficient, such expressions are not uncommon. Since the APLM purports to be an efficient evaluator of expressions, it is worthwhile to look at this example in more detail. The machine code for this expression is:

OP	OPERAND	COMMENTS
LDNF	N	
IOTA		This gives the compressee, ιN
LDNF	N	
IOTA		
LDNF	N	
IOTA		These are the ιN operands of outer product
GDF	MOD	(ιN)∘.|ιN — Matrix of remainders of all possible divisions
LDS	0	
EQ		0=(ιN)∘.|ιN — Has 1 for each 0 remainder, else 0
LDS	1	
RED	ADD	+/[1]0=(ιN)∘.|ιN — Add rows of this matrix
LDS	2	
EQ		2=+/[1]0=(ιN)∘.|ιN — Find which columns have two 1 entries
LDS	1	
CMPRS		Do compression. These are the primes
LDN	PRIMES	Assign result to PRIMES
ASGN		

Since the number of scalar operations performed is the same for both machines, this will not be measured. At the point before executing the LDS 1 instruction which precedes the CMPRS, the state of the APL machine is as shown in Fig. 1.



OP	VALUE	LINK	AUX
IJ	(ιN)		01
RED		8	
IJ	(ιN)		10
IJ	(ιN)		01
GOP	MOD	2	11
IRD	@T1		11
S	0		
OP	EQ	2	11
OP	ADD	7	11
SGV			
S	(-N)		
MIT			
IRD	@T2		01
S	2		
OP	EQ	2	01

FIGURE 1--State of the registers before compress operator.

Up to this point, the NM used memory as follows:

Instruction	Fetches	Stores	Temps	
GDF	N^2+N	$N^2+3N+16$	$N^2+3N+16$	(N+5 stores and temps necessary to evaluate each ιN before GDF + the space for result)
EQ	N^2	N^2	0	
RED	N^2	N+5	N+5	
EQ	N	N	0	
TOTAL	$3N^2+2N$	$2N^2+4N+21$	$N^2+3N+21$	

The count for the APLM at this point is 0 fetches, 9 stores, and 9 temps for the descriptors T1 and T2. However, when the CMPRS operator is found, the left operand must be evaluated as explained in Chapter IV. Thus, the long QS segment must be handed over to the E-machine. This requires N^2+N fetches, N+5 stores, and N+5 temps. In order to do the CMPRS in the NM, the right operand (ιN) must be evaluated, requiring N+5 each of stores and temps. The CMPRS itself takes another N+P fetches, P+5 stores, P+5 temps in the NM, where P is the length of the result. In the APLM, the CMPRS is expanded and deferred, as is the ASGN which follows. The NM requires no work to do the ASGN. The APLM, after this instruction, has its QS full of deferred code for the CMPRS and ASGN. It had to allocate P+5 temps for the result of ASGN (assuming PRIMES was not the correct size already). Passing the QS to the EM requires another N+P fetches and P stores for the APLM. Thus the grand totals are:

	FETCHES	STORES	TEMPS
NAIVE MACHINE	$3N^2$+3N+P	$2N^2$+5N+P+31	N^2+4N+P+31
APL MACHINE	N^2+2N+P	N+P+23	N+P+23

Recall that P is really a function of N, the number of primes less than N, which is asymptotic to $\frac{N}{\log N}$. Thus, we can evaluate the performance ratios between the two machines in some specific cases. These ratios are RF, RS, and RT, the ratios of NM fetches to APLM fetches, stores, and temporaries, respectively. Also of interest is RM, which counts all memory access (fetches + stores), and is the ratio of these two quantities. Table 3 below tabulates these quantities for a few values of N.

TABLE 3

Performance Ratios for Primes Problem as a Function of N

N	P	RF	RS	RM	RT
10	4	2.69	7.7	3.84	4.7
100	25	2.97	138.9	4.91	70.6
500	95	2.99	813.3	4.98	408.0
1000	168	2.997	1683.6	4.99	843.2
5000	669	2.999	8788.8	4.998	4395.8
10000	1229	2.9997	17779.2	4.9992	8891.0
50000	5133	2.99994	90656.6	4.9998	45329.7
$\lim_{N \to \infty}$	$\frac{N}{\log N}$	3	2N	5	N

TABLE 4

Operation Count for One Pass Through Main Loop, Program REC

	NAIVE MACHINE			APL MACHINE		
STATEMENT	FETCHES	STORES	TEMPS	FETCHES	STORES	TEMPS
6	S	2S+5	S+5	0	S+4	4
7	2K	2K+5	K+5	K	K+9	K+9
8	1.5K	0	0	1.5K	0	0
9	8	23	21	8	31	29
10	4S+4	4S+20	2S+20	4S+4	4S+38	2S+38
11	$3S^2+3S$	$2S^2+2S+5$	S^2+S+5	S^2+S	4	4
12	3S+3	3S+8	S+6	S+1	S+9	8
13	$3S^2+9S+1$	$2S^2+6S+22$	$S^2+4S+22$	$2S^2+4S$	$S^2+2S+24$	$S^2+2S+24$
14	$2S^2+2S$	$2S^2+2S+12$	$2S^2+2S+12$	S^2+S	S^2+S+16	S^2+S+16
15	S	S+5	S+5	S	S+9	S+9
TOTAL:	$8S^2+23S+16$ $+3.5K$	$6S^2+20S+105$ $+2K$	$4S^2+12S+101$ $+K$	$4S^2+12S+13$ $+2.5K$	$2S^2+10S+144$ $+K$	$2S^2+6S+141$ $+K$

The above table indicates that the APLM does significantly better than the NM on this program. The RS figures may be deceptive since in terms of total memory accesses the ratio approaches a limit of 5. This is still significant, as is the RT ratio, which increases linearly with N (for large N).

F. Example — Matrix Inversion Programs

As a final example, we analyze the performance of both machines on a standard example, a program which does matrix inversion by elimination with pivoting. To avoid charges of bias, the particular program used was taken from the literature rather than written by the author (Falkoff and Iverson [1968a], p. 19). The program REC is shown in Fig. 2 and has been changed only by altering the syntax of the conditional branch statements. This does not affect the measurements made here and is done purely for esthetic reasons.

Table 4 counts the memory accesses and temporary stores statement-by-statement for one pass through the main loop in program REC. This loop is executed S times. All but the terms involving the variable K are independent of the iteration count. K varies from S to 1 from the first pass to the last. Thus, we can obtain the totals for all passes through the loop by multiplying non-K terms by S and by summing the K terms. This gives the counts in Table 5 below:

TABLE 5

Total Operation Count For Main Loop, Program REC

	FETCHES	STORES	TEMPS
Naive Machine	$8S^3+24.75S^2+17.75S$	$6S^3+21S^2+106S$	$4S^3+12.5S^2+101.5S$
APL Machine	$4S^3+13.25S^2+14.25S$	$2S^3+10.5S^2+144.5S$	$2S^3+6.5S^2+141.5S$

```
          ∇ B ← REC A ; P ; I ; J ; K ; S
             ⍝ MATRIX INVERSION BY ELIMINATION WITH PIVOTING

 1          IF (2=⍴⍴A)∧=/⍴A THEN →L1
              ⍝ ERROR EXIT
 2      L2: ⎕ ← 'NO INVERSE FOUND'
 3          RETURN

              ⍝ S IS DIMENSION OF A
              ⍝ P RECORDS PERMUTATIONS OF ROWS OF A
              ⍝ K SELECTS SUBARRAY OF A FOR ELIMINATION
 4      L1: P ← ⍳K ← S ← 1↑⍴A
              ⍝ ADJOIN NEW COL TO A FOR RESULTS
 5          A ← ((S⍴1),0)\A

              ⍝ ***MAIN LOOP*** (REPEATED S TIMES)
              ⍝ INITIALIZE LAST COLUMN
 6      L3: A[;S+1] ← 1=⍳S
              ⍝ FIND PIVOT ELEMENT, WITH ROW INDEX  I
 7          J ← |A[⍳K;1]
 8          I ← J ⍳ ⌈/J
              ⍝ INTERCHANGE ROWS 1 AND I
              ⍝ RECORD THE INTERCHANGE IN P
 9          P[1,I] ← P[I,1]
10          A[1,I;⍳S] ← A[I,1;⍳S]
              ⍝ CHECK FOR SINGULARITY
11          IF 1E¯30 > |A[1;1] ÷ ⌈/|,A THEN →L2
              ⍝ NORMALIZE PIVOT ROW
12          A[1;] ← A[1;] ÷ A[1;1]
              ⍝ ELIMINATION STEP
13          A ← A-((1≠⍳S) × A[1;]) ∘.× A[1;]
              ⍝ ROTATE A TO PREPARE FOR NEXT STEP
              ⍝ THIS BRINGS 'ACTIVE' SUBARRAY TO UPPER LEFT
14          A ← 1⌽[1]1⌽A
15          P ← 1⌽P
              ⍝ ITERATE ON K
16          IF 0<K←K-1 THEN →L3
              ⍝ DO COLUMN PERMUTATIONS TO PRODUCE RESULT
17          B ← A[;P⍳⍳S]
          ∇
```

FIGURE_2: EXAMPLE PROGRAM: REC

In order to compare the performance of the APL machine to the naive machine, let us form the ratios of the corresponding counts and see how they behave for different values of S. (Recall that S is the dimension of the matrix being inverted by the program under consideration.) The first derivatives of all three ratios are positive for S>0, so that all ratios are increasing as S increases. Table 6 summarizes the properties of the ratios as a function of S.

Let RF(S) by the ratio of fetches in the NM to those in the APLM, RS(S) be the ratio of stores, RT(S) be the ratio of temporary storage allocated, and RM(S) the ratio of all memory accesses (fetches + stores). Then,

$$RF(S) = \frac{8S^2+24.75S+17.75}{4S^2+13.25S+14.25}$$

$$RS(S) = \frac{6S^2+21S+106}{2S^2+10.5S+144.5}$$

$$RM(S) = \frac{14S^2+45.75S+123.5}{6S^2+23.75S+158.75}$$

$$RT(S) = \frac{4S^2+12.5S+101.5}{2S^2+6.5S+141.5}$$

TABLE 6

Machine Comparison Ratios For Main Loop of REC

S	RF(S)	RS(S)	RM(S)	RT(S)
1	1.6	0.847	0.97	0.787
2	1.75	0.99	1.18	0.878
3	1.82	1.15	1.36	0.978
5	1.89	1.46	1.64	1.18
10	1.95	2.04	1.99	1.54
100	1.996	2.94	2.31	1.99
1000	1.9996	2.995	2.332	1.9997
limit $S \rightarrow \infty$	2	3	2 1/3	2

An examination of Table 6 shows that for input arrays A of dimension greater than or equal to 3, 3 the APL machine does better than the naive machine by using fewer fetches and stores. If ρA is 4, 4 or more, fewer temporaries are allocated by the APLM. Finally, the entries for S= 10 and S= 100 show that these improvements rapidly reach the theoretical limits. In the region S≤4 the size of descriptor arrays is approximately the same as the size of the value part of vectors of length S and not much less than the size of arrays of dimension S, S. Thus for small S, the extra overhead in the APLM for creating descriptor arrays in drag-along predominates. However, as S increases, the APL machine improves significantly compared to the naive machine in its economy of memory usage and access.

The program REC used in the previous discussion was taken straight from the literature and was changed only by altering the branch commands and by replacing the operator α by an equivalent construction (because α is no longer a defined operator in APL). Primarily, it is important to emphasize that this is not a specially prepared example designed to tout the virtues of the APL machine. In some sense, this is a "typical" program. By looking more closely at Table 4 we can get a clearer idea of where the APLM does better than the NM and where it lags behind.

Thc APL machine does better (that is, uses fewer fetches, stores, and/or temporaries) than the naive machine on statements 6, 7, 11, 12, 13, 14 does the same as the NM on statement 8, and worse on statements 9, 10, and 15. The places where the NM does better than the APLM are precisely those statements or expressions in which the more successful strategy is to do an immediate evaluation rather than defer the operation. All three are, in this example, statements of the form variable ←T variable, where T is an arbitrary permutation of the subscripts of variable. In all three of these cases, the APLM does worse only by an additive constant, which is the space (and stores) required for a DA to describe the deferred right-hand side of the expression. The NM avoids this by evaluating directly. The same number of fetches are done by both machines for these statements. Of more interest are the cases where the APLM improves on the NM. In all situations these are statements involving more than one operation on the right-hand side of the assignment arrow. By using drag-along and beating, the APLM requires fewer temporaries for intermediate results, which in turn requires fewer stores and consequently fewer fetches when the intermediate results are used later in the expression. The most dramatic demonstration of the efficacy of drag-along is shown in the use of temps in statements 6, 11, and 12 and the stores in statement 11. In all these cases the APL machine uses storage in proportion to the number of array operands while the naive machine requires storage proportional to the size of the array operands. Also, with the exception of statement 10, the number of stores for each statement is proportional to the size of the result for the APLM while in the NM it is generally proportional to both the size of the result and the number of array operations.

As an interesting experiment to see how much these measures of the machine's operation are a function of the actual machine design and how much they depend on the sample program, the author rewrote the function REC in the form shown in Fig. 3, where it is renamed REC1. REC1 is the same algorithm used in REC except that the actual permutations of array A in lines 10 and 14 of REC have been eliminated by using appropriate indexing instead. Also, statement 13 in REC (which corresponds to statement 14 in REC1) is recast to eliminate unnecessary operations and to minimize temporaries in both machines. An analysis of the main loop similar to that for program REC is summarized in Table 7.

```
         ∇ B ← REC1 A ; I ; J ; N ; R ; S ; T ; W
           ⍝ MATRIX INVERSION BY ELIMINATION WITH PIVOTING
           ⍝       'OPTIMIZED' VERSION
           ⍝ THIS PROGRAM DIFFERS FROM REC IN THAT ARRAY
           ⍝ PERMUTATIONS ARE DONE BY CHANGING  THE
           ⍝ PERMUTATION VECTOR, R, RATHER THAN ACTUALLY
           ⍝ PERMUTING THE MAIN ARRAY.  A IS THEN ACCESSED
           ⍝ BY INDEXING WITH R.

 1         IF (2=ρρA)∧=/ρA THEN →L1
 2   L2: ⎕ ← 'NO INVERSE FOUND'
 3       RETURN
 4   L1: R ← ⍳S ← (ρA)[1]
           ⍝ S IS DIMENSION OF A
           ⍝ R RECORDS PERMUTATIONS AND IS USED TO ACCESS A
           ⍝ N COUNTS ITERATIONS
 5       N ← 0
           ⍝ ADD NEW COL TO A; BUILD RESULT IN LEFT COL
 6       A ← (0,Sρ1)\A

           ⍝ ***MAIN LOOP*** (REPEATED S TIMES)
           ⍝ FIND PIVOT ELEMENT
 7   L3: J ← |A[(-N)↓R;N+2]
 8       I ← J ⍳ ⌈/J
           ⍝ INTERCHANGE BY ALTERING PERMUTATION VECTOR
 9       R[1,I] ← R[I,1]
           ⍝ INITIALIZE RESULT COLUMN
10       A[;N+1] ← R[I] = ⍳S
11       IF 1E¯30 > |A[R[1];] ÷ ⌈/|,A THEN →L2
           ⍝ NORMALIZE PIVOT ROW, AND SAVE IN W
12       W ← A[R[1];] ← A[R[1];] ÷ A[R[1];N+2]
           ⍝ T IS ACTIVE COLUMN
13       T ← A[;N+2]
           ⍝ ELIMINATION STEP
14       A[1↓R;] ← A[1↓R;] - T[1↓R] ∘.× W
           ⍝ 'ROTATE' A BY ROTATING R
15       R ← 1⌽R
           ⍝ ITERATE ON N
16       IF S > N←N+1 THEN →L3
17       B ← A[;R⍳⍳S]
         ∇
```

FIGURE 3: 'OPTIMIZED' EXAMPLE PROGRAM: REC1

TABLE 7

Operation Count for One Pass Through Main Loop, Program REC1

	NAIVE MACHINE			APL MACHINE		
STATEMENT	FETCHES	STORES	TEMPS	FETCHES	STORES	TEMPS
7	4S-4N	3S-3N+10	2S-2N+10	2S-2N	S-N+17	S-N+17
8	1.5S-1.5N	0	0	1.5S-1.5N	0	0
9	8	23	21	8	31	29
10	S	2S+5	S+5	0	S+4	4
11	$3S^2+3S$	$2S^2+2S+5$	S^2+S+5	S^2+S	4	4
12	3S+3	3S+8	S+6	S+1	2S+10 *	8 **
13	S	S+5	S+5	S	S+4 *	4 ***
14	$5S^2+5S-10$	$4S^2+4S+19$	$2S^2+4S+26$	$2S^2+4S-6$	S^2+30	31
15	S	S+5	S+5	S	S+9	S+9
TOTAL:	$8S^2+19.5S+1$ $-5.5N$	$6S^2+16S+80$ $-3N$	$3S^2+11S+83$ $-2N$	$3S^2+11.5S+3$ $-3.5N$	$S^2+6S+109$ $-N$ (+10 once)	2S+106 -N (+2S+11 once)

*   +5 once for entire loop
** +S+6 once for entire loop
*** +S+5 once for entire loop

In this algorithm, as in REC, the inner loop is performed S times. The counts shown in Table 7 are independent of the iteration number except for terms involving variable N. Examination of the program shows that N goes from 0 to S-1, increasing by 1 with each pass through the loop. Thus, as in the case of REC, we can obtain total counts for the main loop by summing the N terms and multiplying the others by S. The results are summarized in Table 8.

TABLE 8

Total Operation Counts For Main Loop, Program REC1

	FETCHES	STORES	TEMPS
Naive Machine	$8S^3+16.75S^2+3.75S$	$6S^3+14.5S^2+81.5S$	$3S^3+10S^2+84S$
APL Machine	$3S^3+9.75S^2+4.75S$	$S^3+5.5S^2+109.5S+10$	$1.5S^2+108.5S+11$

An immediate, rather startling observation from this table is that all of its entries are strictly less than the corresponding entries in Table 5 which summarizes the operations of REC. This is somewhat surprising because although the rewriting of the program was done in order to optimize it for the APL machine, it unexpectedly improved performance of the naive machine, as well. In any case, this simply lends more weight to the data summarized in Table 9, where the performance ratios are computed for the two machines operating on this program.

For program REC1, based on the data in Table 8, the ratios are:

$$RF(S) = \frac{8S^2+16.75S+3.75}{3S^2+9.75S+4.75}$$

$$RS(S) = \frac{6S^3+14.5S^2+81.5S}{S^3+5.5S^2+109.5S+10}$$

$$RM(S) = \frac{14S^3+31.25S^2+85.25S}{4S^3+15.25S^2+114.25S+10}$$

$$RT(S) = \frac{3S^3+10S^2+84S}{1.5S^2+108.5S+11}$$

TABLE 9

Machine Comparison Ratios For Main Loop of REC1

S	RF(S)	RS(S)	RM(S)	RT(S)
1	1.63	0.81	0.91	0.8
2	1.91	1.04	1.23	0.99
3	2.07	1.29	1.53	1.21
5	2.24	1.85	2.02	1.77
10	2.41	3.11	2.69	3.88
100	2.64	5.77	3.44	120.2
1000	2.66	5.98	3.49	1871.3
limit $S \rightarrow \infty$	2 2/3	6	3.5	2S

G. Discussion

In the preceding sections we look at a number of typical inputs to the APL machine and find that in all but a few singular cases, it evaluates them more efficiently than a corresponding naive machine. This is a fair kind of comparison because although the naive machine mentioned here is hypothetical, it is based on the design of existing APL implementations, at least one of which is commercially available. The important question, of course, is what kinds of conclusions may we draw from these particular cases? I offer the following:

1. Section D derives lower bounds, all greater than 1, for the ratio between memory accesses and temporary use on the two machines on a simple class of expressions. From this and the previous section it appears that the APLM evaluates expressions of the type analyzed in Chapter II more efficiently than the NM.

2. Operations involving scalar operands are done equally well on both machines.

3. Sections E and F contain more realistic program examples which were analyzed in detail. In both cases, the APLM improves significantly on the NM in its use of memory.

4. The only cases where the APLM does worse are those expressions containing a single operator which does not fit into the beating scheme, and for which the best evaluation strategy is to evaluate immediately, rather than to defer. In these cases, the NM does slightly better than the APLM but only by a small additive constant. (This being the space and stores for the APLM to construct a deferred descriptor.)

In view of the above, it is clear that in most cases, the APL machine design proposed here is more efficient than a naive machine in the sense that for any given program, the APLM uses fewer fetches, stores, and allocates fewer temporaries than the naive machine.*

*A corollary worth noting is that there exist inputs (i.e., programs) for which the APLM always performs worse than the NM according to the measures derived here. However, this should be neither startling nor alarming and does not detract from the general conclusion above.

CHAPTER VI

CONCLUSIONS

In this chapter, we will summarize all that has gone before and indicate some directions for future research on this subject.

A. Summary

Although the original goal of this investigation was to produce a machine architecture appropriate to the language APL, some of the work done in pursuit of this goal is intrinsically interesting in itself. In particular, we call attention to the mathematical analysis discussed in Chapter II. In Chapter II, we find that there is a subset of APL operators (the selection operators) whose compositions are also selection operators. Further, compositions of these operators can be represented compactly in a standard form. Moreover, there is a set of transformations sufficient to transform any expression consisting solely of selection operators acting on a single array into an equivalent expression in standard form. By extension, similar results are described that apply to select expressions which include scalar arithmetic operators, reductions, and inner and outer products.

One result, of at least theoretical interest, is that all inner products can be represented as a reduction of a transpose of an outer product (Theorem Tb). The general dyadic form is introduced in Chapter II as a vehicle for extending the results about selection operators on single arrays or scalar products to analogous results on inner and outer products.

In Chapter III, we show that if arrays are represented in row-major order and if the representation of the storage access function for an array is kept separate from the array value, then the result of applying a selection operator to an array can be obtained simply by transforming the mapping function. This approach is the basis for beating, one of the novel features of the APL machine. In mathematical terms, beating is equivalent to the following: if an array is construed as a function (the storage access function S) applied to an ordered set of values A, and if F1, F2, ..., FN are selection operators then the sequence

$$F1(F2(\ldots(FN(S(A)))))$$

is equivalent to some new function T(A) where T is a functional composition with o:

$$T \longleftarrow (F1 \circ (F2 \circ (\ldots \circ (FN \circ S)))) \quad .$$

Chapter IV describes a machine based on the beating process and the drag-along principle. The latter says that all array calculations should be deferred as long as possible in order to gain a wider context of information about the expression being calculated. This is done because of the possibility that extra information might allow the simplification of the expression to be evaluated. This is particularly important when, as in APL, operands are array-shaped. In effect, a language like APL which allows sophisticated operations on structured data to be encoded very compactly, makes it possible to write expressions which, though innocent-looking, require much calculation. In fact, one major goal of the machine design is to minimize any unnecessary calculations in evaluating APL programs. Thus, drag-along becomes an important way of doing so. Drag-along combines all element-by-element operations in an incoming expression into a single, more complex, element-by-element operation which need only be done once for each element of the result array. This is based on the fact that for most APL operators, $\underline{F}$,

$$A\ \underline{F}\ B \text{ means for all } L\ \underline{ELT}\,\iota\rho(A\ \underline{F}\ B)$$

$$(A\ \underline{F}\ B)[;/L] \leftrightarrow (\underline{F1}\ A)[;/L]\ \underline{F}\ (\underline{F2}\ B)[;/L],$$

where $\underline{F1}$ and $\underline{F2}$ depend on $\underline{F}$ and are normally the identity function. Simply stated, this says that a single element of an array-valued expression can be computed by evaluating a similar expression of single elements.

The APL Machine is divided into two submachines, the Deferral Machine and the Execution Machine, in order to facilitate drag-along and beating. Conceptually, the DM is a dynamic, data-dependent compiler which examines incoming expressions (machine code) and their operand values (data) and produces instructions to be executed by the EM. This code is deferred in an instruction buffer and can also be operated upon by the DM. At appropriate times, control is passed to the EM which executes the deferred instructions. Since EM code must compute an array-valued result, a stack of iteration counters are used by the E-machine to produce all elements of the result one at a time. A feature of the APLM which makes it easy for the DM to manipulate its own deferred code is that programs (and deferred code) are organized into segments which contain only relative addresses. Thus pieces of program can be referenced by descriptors, and these pieces can be relocated at will simply by changing the descriptors and not the code. This scheme leads to the use of a stack of instruction counters, each one of which refers to a currently active segment in either the EM or the DM. Thus it is easy for the machine to change state and recover previous states, thereby simplifying the entire control process.

Chapter V contains a discussion of the machine design in which it is shown that at worst, the APL Machine performs the same as a naive machine executing the same program and at best shows a significant improvement. The primary parameters used in the evaluation are measures of memory utilization. Other measures, such as encoding densities, are not appropriate,as this aspect of the machine design has not been specified. Such measures should be taken into account, however, if it is desired to implement a machine such as this. The evaluation of a subset of APL containing only scalar arithmetic operators and select operators shows that the APLM approaches the theoretical minimum of memory accesses and temporary storage utilization for this class. Further, the ratio of accessing operations between the NM and the APLM are significant since the NM expends effort for fetching and storing in proportion to the number of operators in an expression while the APLM does fetches in proportion to the number of operands and stores only once. Similarly, it is noted that for this class of expressions, the APLM needs to allocate space only for the result of an expression while the NM requires temporary storage which is a function of the tree structure of the expression being evaluated.

In the same chapter, an analysis of an APL "one-liner" and a matrix inversion program containing a more general mix of operators, shows that the APLM does better than the NM by at least a factor of 2 on these measures. A final observation is that the APLM described here is not significantly different in complexity from a naive machine. Thus, it could presumably be implemented with approximately the same resources. Hence, it appears that this design is an improvement and could profitably be used in future incarnations of machines for APL.

Although the APL machine is an improvement over the naive approach, it would be absurd to claim that it is the "final solution" to the problem. Clearly, it is not. There are still some functions, such as compression or catenation, which it handles awkwardly. Similarly, it is distasteful (and inefficient) to evaluate operands of a GDF explicitly if they are other than simple select expressions. Ideally, there should be no temporary storage used for the evaluation of expressions without side effects (such as embedded assignment). Thus, there is still work to be done on this problem.

B. Future Research

The ideas summarized here tend to fall into two classes — extensions or refinements of the work already reported, and new problems suggested by the current research.

In the second category is the area of mathematical analysis of APL operators. The work in Chapter II of this dissertation barely skims the surface of this topic. The general problem, of course, is at the heart of "Computer Science," namely the study of data-structures and operations upon them. However, APL and its extensions are rich in mathematical interest and this field deserves further, more concentrated investigation. Similarly, the results found in Chapter II as well as the structure of the machine have implications for language design. An important next step is to take some of the ideas which appear in the machine or the analysis and attempt to map them back into the programming language. As a trivial example, the ease with which the machine evaluates select expressions suggests that there ought to be the possibility of more general select expressions allowed to the left of an assignment arrow, e.g., it should be possible to say (1 1⍉*M*)←*A*, meaning assign *A* to the main diagonal of *M*. Again, the ease with which the APLM does segment activation suggests that there should be some parallel facility in a programming language. At the very least, APL should contain some more sophisticated sequence-controlling operations such as case, conditional, and repeat constructs. A final possibility along these lines is suggested by the similarity among the various selection operations. Simply that there exists such a compact standard form suggests that there might be a different, perhaps more general, set of selection primitives which would be desirable in a language like APL.

In the direction of refinements there are several areas of interest. One is to try to add more parallelism to the machine. In this work, we have used the implied parallelism of APL in drag-along and beating, but it appears not to be fully exploited. For instance, there is the interesting possibility of making the DM and the EM more independent, thus gaining an amount of parallelism. There is no reason, for example, why there could not be multiple copies of both, working simultaneously on different parts of an expression or program. Another place where parallelism could be exploited is in the E-machine. Instead of doing everything in serial, much could possibly be done on a grander scale.

It appears possible to extend the formulation of the standard form to include more operators such as catenation, restructuring, rotation, compression, expansion, and explicit indexing. If such a general form could be found, the operation of the machine could be simplified and perhaps made more efficient.

In order to have any real implementation of the machine, it will have to be extended to include instructions for input and output and other systems-type functions. Also, as soon as an implementation is attempted, problems such as encoding of data and instructions will have to be broached. Similarly, it will probably be necessary to consider the question of data types in a real incarnation of the APL machine. Other machine extensions which might be considered is the addition of a set of registers (possibly stacks) for eliminating some of the problems of temporary storage in EM code which does not follow the stacking discipline of VS. This, in turn, entails the addition of instructions to the machine's repertoire, although these might not have to be visible to the programmer.

Although on the one hand it is counter to the idea of a language-oriented machine, it might be desirable to give the (systems) programmer more direct control over the E-machine. In particular, this would make it possible to "pre-compile" particular segments for the EM when enough information is available in advance. An interesting extension of this is to allow the EM to call upon the DM in the same way that the DM uses the EM. This would make the overall system more symmetric and might increase its power and versatility.

A further area of investigation combines language and machine design. This is the problem of extending APL to include more general kinds of data structures, such as lists or records, and then attempting to fit these into the structure of the machine. This problem, in turn, makes further demands on the mathematical analysis of the language and its operators.

Finally, it is important to investigate the possibility of extending some of the methods and results of this work to other languages and data structures.

C. Concluding Remarks

This chapter has summarized the mathematical analysis and machine design reported in this dissertation and has indicated some directions for fruitful investigations in the future. It is pleasing to be able to end this work with a feeling of accomplishment, yet it is perhaps more satisfying to know that this is not really an ending, but a beginning.

The Road goes ever on and on,
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with weary feet,
Until it meets some larger way,
Where many paths and errands meet.
And whither then?...
I can not say.

J.R.R.Tolkien

REFERENCES

Abrams, P. S. [1966]. An Interpreter for "Iverson Notation." Report No. CS47, Computer Science Department, Stanford University (August 17).

Adams, D. A. [1968]. A Computation Model with Data Flow Sequencing. Report No. CS117, Computer Science Department, Stanford University (December).

Amdahl, G. M. [1964]. The Structure of SYSTEM/360: Part III — Processing Unit Design Considerations. IBM Systems Journal, Vol. 3, No. 2, 144-164.

Amdahl, G. M., Blaauw, G. A. and Brooks, Jr., F. P. [1964a]. Architecture of the IBM SYSTEM/360. IBM Journal of Research and Development, Vol. 8, No. 2 (April), 87-101.

Anderson, J. P. [1961]. A Computer for Direct Execution of Algorithmic Languages. 1961 Eastern Joint Computer Conference, The Macmillan Company, New York, 184-193.

Bairstow, J. N. [1969]. Mr. Iverson's Language and How It Grew. Computer Decisions, Vol. 1, No. 1 (September), 42-45.

Barton, R. S. [1961]. A New Approach to the Functional Design of a Digital Computer. Proceedings of the Western Joint Computer Conference, 393-396.

Barton, R. S. [1965]. The Interrelation Between Programming Languages and Machine Organization. Proceedings of the IFIP Congress 1965, Vol. 2, 617-618.

Bashkow, T. R. [1964]. A Sequential Circuit for Algebraic Statement Translation. IEEE Transactions on Electronic Computers, Vol. EC 13 (April), 102-105.

Bashkow, T. R., Sasson, A. and Kronfeld, A. [1967]. System Design of a FORTRAN Machine. IEEE Transactions on Electronic Computers, Vol. EC-16, No. 4 (August), 485-499.

Bayer, R. and Witzgall, C. [1968]. A Data Structure Calculus for Matrices. Report No. 20, Information Sciences Laboratory, Boeing Scientific Research Laboratories, Seattle, Washington, (May).

Berry, P. [1969]. APL\360 Primer. Form No. C20-1702-0, International Business Machines Corp., White Plains, New York.

Berry, P. [1968]. APL\1130 Primer. Form No. C20-1697-0, International Business Machines Corp., White Plains, New York.

Branin, Jr., F. H., Hall, L. V., Suez, J., Carlitz, R. M., and Chen, T. C. [1965]. An Interpretive Program for Matrix Arithmetic. IBM Systems Journal, Vol. 4, No. 1, 2-24.

Breed, L. M. and Lathwell, R. H. [1968]. The Implementation of APL\360. Interactive Systems for Applied Mathematics, Academic Press, New York, 390-399.

Buchholz, W. [1962]. Planning a Computer System, McGraw-Hill Book Co., New York.

Burks, A. W., Warren, D. W. and Wright, J. B. [1954]. An Analysis of a Logical Machine Using Parenthesis-Free Notation. Mathematical Tables and Other Aids to Computation, Vol. 8, No. 46 (April), 53-57.

Burroughs Corporation [1963]. The Operational Characteristics of the Processors for the Burroughs B5000. Burroughs Corporation, Detroit, Michigan.

Clark, E. R. [1967]. On the Automatic Simplification of Source-Language Programs. Communications of the ACM, Vol. 10, No. 3 (March), 160-165.

Cohen, J. [1967]. A Use of Fast and Slow Memories in List-Processing Languages. Communications of the ACM, Vol. 10, No. 2 (February), 82-86.

Collins, G. E. [1965]. Refco III, A Reference Count List Processing System for the IBM 7079. Research Report No. RC-1436, IBM Research Division, Yorktown Heights, New York (May 11).

Davis, G. M. [1960]. The English Electric KDF9 Computer System. Computer Bulletin, Vol. 4, 119-120.

Dijkstra, E. W. [1968]. Go To Statement Considered Harmful. (letter) Communications of the ACM, Vol. 11, No. 3 (March), 147-148.

Elspas, B., Goldberg, J., Green, M., Kautz, W. H., Levitt, K. N., Pease, M. C., Short, R. A., and Stone, H. S. [1966]. Investigation of Propagation-Limited Computer Networks. Report No. AFCRL 64-376 (III). Stanford Research Institute, Menlo Park, California (June).

Falkoff, A. D. [1965]. Formal Description of Processes — The First Step in Design Automation. Research Note No. NC-510, IBM T. J. Watson Research Center, Yorktown Heights, New York (June).

Falkoff, A. D. and Iverson, K. E. [1968a]. The APL\360 Terminal System. Interactive Systems for Applied Mathematics, Academic Press, New York, 22-37.

Falkoff, A. D. and Iverson, K. E. [1968b]. APL\360: User's Manual. International Business Machines Corp., Yorktown Heights, New York (July).

Falkoff, A. D., Iverson, K. E. and Sussenguth, E. H. [1964]. A Formal Description of SYSTEM/360. IBM Systems Journal, Vol. 3, No. 3, 198-262, (Errata; Ibid., Vol. 4, No. 1, 84).

Galler, B. A. and Perlis, A. J. [1962]. Compiling Matrix Operations. Communications of the ACM, Vol. 5, No. 12 (December), 590-594.

Galler, B. A. and Perlis, A. J. [1967]. A Proposal for Definitions in ALGOL. Communications of the ACM, Vol. 10, No. 4 (April), 204-219.

Hellerman, H. [1964]. Experimental Personalized Array Translator System. Communications of the ACM, Vol. 7, No. 7 (July), 433-438.

Hill, U., Langmaack, H., Schwarz, H. R. and Seegmüller, G. [1962]. Efficient Handling of Subscripted Variables in ALGOL 60 Compilers. Proceedings of 1962 Rome Symposium on Symbolic Languages in Data Processing, Gordon and Breach, New York, 331-340.

Hillegass, J. R. [1968]. Burroughs Dares to Differ. Data Processing Magazine (July).

Hoare, C.A.R. [1968]. Subscript Optimization and Subscript Checking. Algol Bulletin, No. 29 (November) 33-44.

Iliffe, J. K. [1968]. Basic Machine Principles. American Elsevier Publishing Company, New York.

Iliffe, J. K. and Jodeit, J. G. [1962]. A Dynamic Storage Allocation System. Computer Journal, Vol. 5, 200-209.

Iverson, K. E. [1966]. Elementary Functions: An Algorithmic Approach, Science Research Associates, Inc., Chicago, Illinois.

Iverson, K. E. [1964]. Formalism in Programming Languages. Communications of the ACM, Vol. 7, No. 2 (February), 80-88.

Iverson, K. E. [1962]. A Programming Language, John Wiley and Sons, New York (1962).

Iverson, K. E. [1963]. Programming Notation in Systems Design. IBM Systems Journal, Vol. 2, No. 2 (June), 117-128.

Jodeit, J. G. [1968]. Storage Organization in Programming Systems. Communications of the ACM, Vol. 11, No. 11 (November), 741-746.

Knuth, D. E. [1967]. The Remaining Trouble Spots in ALGOL 60. Communications of the ACM, Vol. 10, No. 10 (October), 611-618.

Knuth, D. E. [1968]. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison Wesley, Reading, Massachusetts.

Korfhage, R. R. [1965]. Deeply Nested Iterations. Communications of the ACM, Vol. 8, No. 6 (June), 377-378.

Lawson, H. W. [1968]. Programming-Language-Oriented Instruction Streams. IEEE Transactions on Computers, Vol. C17, No. 5 (May), 476-485.

Lesser, V. R. [1969]. A Multi-Level Mirco Computer Architecture. Report No. CGTM-87 , Stanford Linear Accelerator Center, Stanford University, Stanford, California.

Lowry, E. S. and Medlock, C. W. [1969]. Object Code Optimization. Communications of the ACM, Vol. 12, No. 1 (January), 13-23.

McCarthy, J. [1963]. A Basis for a Mathematical Theory of Computation. Braffort, P. and Hirschberg, D. (eds.), Computer Programming and Formal Systems, North-Holland Publishing Co., Amsterdam, The Netherlands.

McCarthy, J. [1966]. A Formal Description of a Subset of ALGOL. Steel, Jr., T. B. (ed.), Formal Language Description Languages for Computer Programming, North-Holland Publishing Co., Amsterdam, The Netherlands, 1-7.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. [1962a]. Lisp 1.5 Programmer's Manual. MIT Press, Cambridge, Massachusetts.

McCarthy, J. [1962b]. Towards a Mathematical Science of Computation. Proceedings of the IFIP Congress 1962, North-Holland Publishing Co., Amsterdam, The Netherlands.

McKeeman, W. M. [1966]. An Approach to Computer Language Design. Report No. CS48, Computer Science Department, Stanford University, (August 31).

McKeeman, W. M. [1967]. Language Directed Computer Design. 1967 Fall Joint Computer Conference, Thompson Books, Washington, D. C., 413-417.

Meggitt, J. E. [1964]. A Character Computer for High-Level Language Interpretation. IBM Systems Journal, Vol. 3, No. 1, 68-78.

Melbourne, A. J. and Pugmire, J. M. [1965]. A Small Computer for the Direct Processing of FORTRAN Statements. The Computer Journal, Vol. 8 (April), 24-28.

Mendelson, E. [1965]. Introduction to Mathematical Logic, D. Van Nostrand Co., Princeton, New Jersey.

Mikhnovskiy, S. D. [1965a]. Addressing of Elements of a Block Using Address Scales. Glushkov, V. M. (ed.), Problems in Theoretical Cybernetics, Naukova Dumka Publishing House, Kiev, U.S.S.R. Translation: JPRS Washington, D. C. (1966), 71-80.

Mikhnovskiy, S. D. [1965b]. A Method for Abbreviated Notation of Blocks of Data. Ibid., 38-44.

Mullery, A. P., Schauer, R. F. and Rice, R. [1963]. ADAM: A Problem Oriented Symbol Processor. 1963 Spring Joint Computer Conference, Spartan Books, Washington, D. C., 367-380.

Myamlin, A. N. and Smirnov, V. K. [1968]. Computer with Stack Memory. IFIP Congress 68, D91-D96.

Naur, P. (ed), [1963]. Revised Report on the Algorithmic Language ALGOL 60. Communications of the ACM, Vol. 6, No. 1 (January), 1-17.

Pakin, S. [1968]. APL\360 Reference Manual, Science Research Associates, Inc., Chicago, Illinois.

Randell, B. and Russell, L. J. [1964]. ALGOL 60 Implementation, Academic Press, London.

Satterthwaite, E. [1969]. MUTANT 0.5, An Experimental Programming Language. Report No. CS120, Computer Science Department, Stanford University, Stanford, California (February 17).

Sattley, K. [1961]. Allocation of Storage for Arrays in ALGOL 60. Communications of the ACM, Vol. 4, No. 1 (January) 60-65.

Senzig, D. N. and Smith, R. V. [1965]. Computer Organization for Array Processing. 1965 Fall Joint Computer Conference, Spartan Books, Washington, D. C., 117-128.

Sugimoto, M. [1969]. PL/I Reducer and Direct Processor. Proceedings of the 24th National Conference, Association for Computing Machinery, New York.

Wagner, R. A. [1968]. Some Techniques for Algorithm Optimization with Application to Matrix Arithmetic Expressions. Computer Science Department, Carnegie-Mellon University (June 27).

Weber, H. [1967]. A Microprogrammed Implementation of EULER on IBM System/360 Model 30. Communications of the ACM, Vol. 10, No. 9 (September), 549-558.

Wilkes, M. V. [1965]. Slave Memories and Dynamic Storage Allocation. IEEE Transactions on Electronic Computers, Vol. EC-14, No. 2 (August), 270-271.

Wirth, N. [1967]. On Certain Basic Concepts of Programming Languages. Report No. CS65, Computer Science Department, Stanford University, Stanford, California (May 1).

Wirth, N. and Weber, H. [1966]. EULER: A Generalization of ALGOL and its Formal Definition,

Part I: Communications of the ACM, Vol. 9, No. 1 (January), 13-23;

Part II: Communications of the ACM, Vol. 9, No. 2 (February), 89-99;

Errata: Communications of the ACM, Vol. 9, No. 12 (December), 878.

Wortman, D. W. [1970]. PL/I Directed Language Design (to appear).

Yershov, A. P., Kozhokhin, G. I. and Volushin, U. M. [1963]. Input Language For Automatic Programming Systems, Academic Press, London.