UIT Report Vol. 12, No. 2, pp.239~248, 1981 연구논문집 제12권 제2호 pp.239~248, 1981

A Study on APL Interpreter Using Ladder Mechanism. *

Chisu wu
Dept. of Computer Science
(Recieved April 10, 1981)

〈**Abstract**〉

APL implementations have been carried out by transformations of array accessings, reordering of APL operators, syntactical analysis and using APL idioms.

In this study, I propose that A-Machine code is generated by a translator which is written by APPLE-PASCAL using ladder mechanism.

Ladder를 이용한 APL Interpreter 구성에 관한 연구

우 치 수
전 자 계 산 학 과
(1981. 4. 10 접수)

〈**적 요**〉

APL의 보완은 배열의 접근방식의 변환, 연산자의 재배치, 문법분석, APL의 숙어의 사용등에 의하여 행하여졌다.

본 논문은 배열의 접근방식을 위한 LADDER를 사용하여 APPLE-PASCAL을 이용한 번역기에 의한 A-Machine를 구성하는 방법의 제시이다.

Ⅰ. Introduction

1. A Programming Language

A progamming language is based on the work of K.E. Iverson.[3] APL is concise, highly mathematical language designed to deal with array-structured data. APL program generally contain expression with arrays as operands and which evaluate to arrays, while most other languages require that array manipulations to be expressed element by element. To faciliate its use of array as operand, APL has rich operators for array calculations. Also it is highly consistent internally both synatactically and semantically, and hence could be called mathematical. Because of its use of its use of structured data and its primitives which are quite different from those class of classical digital computer, APL does not fit well onto ordinary machines. It is possible to do so, and interpreter have been written at least five different machines. Finally, because of its mathematical properties, it is possible to discuss the semantics of the language rigorously and to derive significant formal results about expression in the language.

2. The Problem

In this thesis we represent a design of system for excuting the language which alleviate some

* 이 연구는 1980년 문교부 학술 연구조성비에 의하여 연구된 것임.

of the difficulties that have been cited as limiting the continued growth of the language. In particular we address the following problems.

1. APL function will often generate large arrays as as intermediate result.

2. APL's interpretative execution can be slow compared to that attained with programs compiled from languages such as FOTRAN or Algol 60.[4]

3. The style of APL programming most efficient for experienced user tends to worsen the problem 2.

3. Previous Work

Implementation of APL have been attempted to solve problems mentioned 1.2.

(1) Simple Interpreter

The original implementations of APL and that followed were interpreters which execute each operator separately as encounted. All intermediate results are stored in memory. Great speed improvement has been obtained by fine tuning of the routines for various operators and by the recognition of short special patterns of operators which are often called APL's idioms. However, the reduction of temporary storage may require interleaving the individual calculations of a sequences of operations. It is clear that the sequences of operations which can be profitably interleaved are too long to be recognized as special cases.[5]

(2) Translation to Algol

A transltor which translate subset of APL to Algol was implemented by Jekins[4] He was forced to restrict the language so that compilation could take place before any data was available, and so a compiled module would always remain valid. The feature of APL which present difficulty in that regard are discussed later. While compiled code is significantly faster than interpreted APL for scalar calculations the advantage almost disappears for large array. Jekins did not investigate the reasons for the inefficiency of array calculation. However experience gained in this implemention suggest that it is resulted from sequential execution of operators which require large amounts of temporary storage, and from the cost of array element address generation.

Ⅱ. Array Access

1. Address-sequencing Technique

Data selection in random access memories is accomplished by means of an address, a number that uniquely identifies a cell in the memory. To step through the data elements in an APL array in a particular order, the element must be stored in a random access memory, yields the desired sequence of data elements. Since one iteration of this algorithm is executed each time a data element is fetched, the time required for the algorithm to generate one address from previous one should be short as possible. The arithemetic technique uses the operators $+, -, *$ and/in the sequencing algorithm. The expression that yields the location of an element of FOTRAN array from its subscription and dimension is:

$$\begin{aligned} LOC(A(i_1,i_2,i_3)) &= LOC(A(1,1,1))+i_1-1+ \\ &\quad k(i_2-1+k_2(i_3-1)) \\ &= LOC(A(1,1,1))+\sum_{j=3}^{3} \\ &\quad (i_j-1)\prod_{l=1}^{j-1} k_l \end{aligned}$$

LOC(X) is the location of X and A is dimensioned $A(k_1, k_2, k_3)$. The code generated by some compilers can reference array elements with less work than is reqired to evaluate this expression; one method reference an array element with no multiplication. It is not necessary to subtract from all subscripts if LOC($A(0,0,0)$) is substitute for $LOC(A(1,1,1))$ in expressions above; while $A(0,0,0)$ does not exist in FORTRAN, this imaginary location

implifies the address-generation expression:

$$LOC(A(i_1, i_2, i_3))=LOC(A(0,0,0))+i_1+k_1$$

$$(i_2+k_2 i_3)=LOC(A(0,0,0)+\sum_{I=1}^{3}(i_j\prod_{I=1}^{I=1}k_1)$$

The evaluation of the first of the first of these two simplified expressions for N-dimensional array reqirees $N-1$ multiplications and N additions. For $A \leftarrow B+C$, this is a tremendous amount of overhead to insert at each step. But the FORTRAN address-generation expression produces an address given the subscripts of the desired element; the address generation expression for the simple APL statement must sequence through all the addresses and need not be able to select elements in arbitary order with the same ease.

An optimizing FORTRAN compiler would use a more efficient method than the expression given abovess elemsent of an array, especially if the references occurred within the range of the *DO* loop where the index variable of the *DO* loop was one of the subscripts of the array reference.

2. Ladders

A ladder is a structure, introduced by Perlis[6], that combines a data acces mechanism for array structured data with computational and linkage capability. One ladder is associated with each occurance of an array in an APL statement are connected to from a ladder network for the statement.

A ladder can be divided into two part, a fixed part are a variable part. The fixed part handles the accessing of the array elements and check for completion. It is associated. The variable part performs the computation specified by the operators in the statement.

The fixed frxmework ladders provide for an APL statement to be generated easily. The fixed parts of the ladders for an APL statement can be formed as soon as the ranks of arrays referenced by the statement are known. If the ranks are not known at the time the statement is to be executed then the statement must be broken into parts. The first part is the largest part, beginning at the right of the statement, for which the ranks of all the arrays are known. The ladder network for this statement fragment can be formed and executed. After the execution another ladder network is formed from the rest of the statement in the same way as before. Then the network is executed. This process continous until the entire statement has been executed. Any legal APL statement can be executed in this manner.

The variable parts of each ladder are generated in much the same way code is generated by a standard compiler. If the ladder network for a ststement is saved after the statement has been executed, it is likely that the same network can be used the next time the statement is encountered. The conditions in which a ladder network can be reused are discussed in more detail below.

Ladders can be explained most effectively by first discussing the workings of the data-access mechanism. The mechanism used in similar to others[1].

Let A be an array of rank three and shape k_1, k_2, k_3 stored in ravel(row-major) order. the separation between adjacent elements on the same row(differing in only the last subscript) is 1. The separation between elements differing only in the second subscript(and there only by one) is k_3, or number of elements in a row. And the separation between elements differing by one in the first subscript and not at all in the others is $k_2 \times k_3$, or the number of elements in a planar cross-section of the array. The storage arrangement is the same as the one used for storage of FORTRAN array except that the order of the subscripts is reversed. The analogous expression that gives the location of a paricular element of the array is:

$$LOC(A(i_1 : i_2 : i_3))=LOC(A(1:1:1))+i_3-1$$

$$+k_2(i_2-1)+k_2k_3(i_1-1)$$
$$=LOC(A(1;1;1))+\sum_{j=1}^{3}((i_1-1)\prod_{l=j+1}^{3}k_l)$$

Consider the problemof delivering, in ravel order, the elements of the array A to a computational procedure. The address of the element to be delivered must be initialized to the address of the element to be delivered must be initialized to the address of the element. Since the array is stored in the row may be produced by adding one to the current address. Continuing to apply this procedure will yield the sequence of addresses of all the data elements in the first row. Because rows are stored sequentially, the last element of one row is followed immediately by the first element of the next row; therefore, adding one to the address of the last element of one row will generate the address of the first element in the next. In fact, adding one will always produce the address of one data element from the address of the previous data element. This procedure obviously work; it is clear that beginning with the address of the first element of an array stored in ravel order and adding one to that address until the address of the last element in the array is reached will produce the sequence of addresses of the data elements in ravel order. A simple extension of this technique can produce the sequence of addresses of the elements of an array in ravel order even though the array itself is not stored in ravel order.

The mechanism depicted in the figure. 1 is designed to produce the addresses of the elements of an array in ravel other. In the figure, PI is a pointer that is initialized and stepped so that it sequences through the elements in the array referenced by the ladder in ravel order. BETA represents the address of the first element in the array, the array's base address. The RHO variables are the components of the shape, or dimension vectors of the array and the I variables are the subscripts. The origin is assumed to be one. When control passes through the bottom block, PI points to the array element specified by the subscripts.

It is clear that the mechanism will step through the elements of an array stored in ravel order ($DELTA_1=DELTA_2=DELTA_3=1$). You will see that the application of certain common APL operators to an array stored to permit access in ravel order by this mechanism yields an array that can be accessed in ravel order by the same mechanism but with different BETA, RHO, and DELTA. Since these operators change the logical order but not the physical order of the data, the resulting array is not stored in ravel order. The mechanism can be used to access these arrays, so storage orderings that the mechanism can handle that differ from the ravel ordering are common. The operators, which Abrams called selection operators, are reverse, transpose, take, drop and certain types of subscriptings.

It is sometimes necessary to access a particular element of an array, as specified by its subscrpts. For an array that the procedure descibed above can access. The address of a particular element of an array is given by:

$$LOC(A(j_1;j_2;\cdots j_n))=\text{BETA}+\sum_{i=1}^{N}(j_i-1)G_i$$

Not only does the mechanism sequence through the addresses and perform a check for completion, but it is also enables some computations to be simplified. The question remains how to interface this structure with the computation still reqired bo the statement.

Figure. 2 shows a ladder structure for an array of rank three as described above except that seven numbered boxes, called splices, have been inserted in the structure. Code to perform the calculations associated with the statement is placed in the splices. For an APL statement involving only the simple scalar

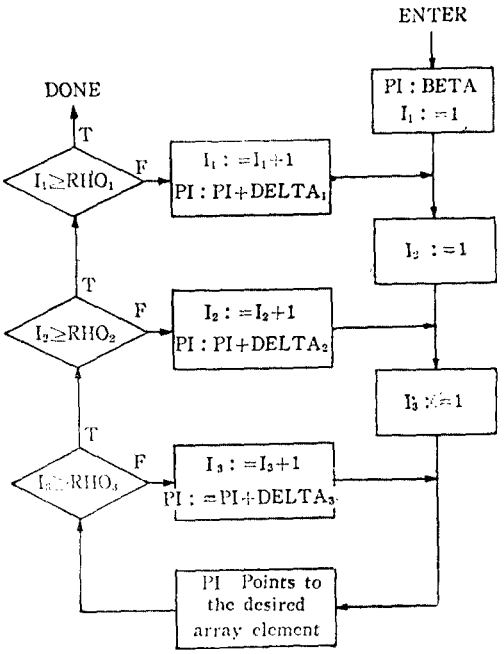

Fig. 1 Data-access mechanism

operators, all code goes into the splice in the innermost loop, splice 4 the rank-three case. In addition one of the arrays has an END instruction in the splice encountered only when an entire pass has been made through the array, splice 7 for the rank-three example. The additional splices are used for the more complicated operators like inner or outer product, scan, and reduction. The instruction in the splices are simple scalar oriented instructions; they include the basic arithmetic operations, some of the Boolean operations, and testing and branching instructions.

The ladder structure provides a way to step through the elements in an array and to perform computations on them. Since almost all APL statements involve more than one array, there must be a way to link ladders so that data as well as control can be transferred among them. The data can be transferred by allowing the cede in each ladder to reference a shared memory where all data (except the arrays themselves) are stored. The binding of the code to this shared data take place at code-generation time. Control is passed among ladders by means of coroutine jumps.

Most operators in APL can be accomodated within the computational structured outlined above. The fixed address sequencing that the ladder mechanism provides is not suitable for some APL operators such as grade up and down; such operators can be handled by standard interpretation technique.

The ladder structure handles changes in the shape of arrays automatically. Once the ladder and the associated splice code have been generated, they can be used the next time statement is executed if the conditions described by Perlis[5], are not violated. The rank and type of each variable in the statement may not change from execution to execution; every

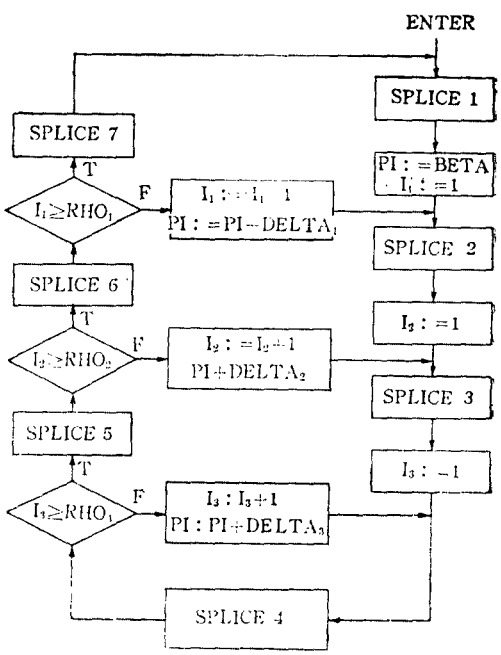

Fig. 2 Ladder mechanism

function called in the statement must return a result with the same rank and type for every call provided the same arguments also have the same rank and type; and no $(X)\phi Q$ or $(X)\phi Q$ may occur where X is an expression other than a constant vector and Q is any expression. Of course, a perfectly legal APL program can violate these conditions, and, as a result, the ladders and splices code will have to the regenerated.

These limits on the use of ladders in the execution of APL programs detract from the elegance of the approach, but the thrust of this work is to improve the overall efficiency of execution of APL programs even at the expense of some elegance.

In summary, ladders provide a data-access mechanism for arrays in APL statements. Some APL operators can be implemented by altering data-access parameters without fetching any elements of the array argument; most others can be executed by inserting code into the splices in the strucure. For most APL statements the ladders and splice code generated when each statement is first executed can be interpretation/compilation overhead. A normal interpretation scheme can be used for for those APL statements that certain operands or operators not easily implemented with ladder, such as null array and grade up/down. Since, for som APL statements, accessing array elements requires a large fraction of the statement's execution time, an efficient accessing mechanism will allow the construction of a fast APL processor.

Ⅲ. A-Machine

To implement programmimg languages such as FORTRAN, BASIC or PASCAL, sometimes special machines are devised as FORTRAN-Machine, BASIC-Machine, PASCAL-Machine. According to programming language, each language has special features: APL has many operators to manipulate mathematical arrays, PASCAL has some keywords to handle files.

Generally, host machine has compiler and it makes equivalent code for special machine for given program and execution is handled by special machine. For examples, P-machine is used to implement APPLE-II PASCAL[2].

Now I am going to propose the A-machine which has instructions in Table 1. Those are appropriate to execute APL scalr operators.

Selection operators are manipuated by host machines program (compiler) using array accessing mechanism. By the way selection operators is done by reorganized accessing at the run time.

A-machine's instructions are divided four classes accoring to APL operators.

Table. 1. Single Operand Instruction

		OP	OPN
INC	0000001	000001	
DEC	0000001	000010	
TST	0000001	000011	
CLR	0000001	000100	
NEG	0000001	000101	
NOT	0000001	000110	
RECIP	0000001	000111	
SIG	0000001	001000	
CEL	0000001	001001	
FLR	0000001	001010	
ABS	0000001	001011	
ROLL	0000001	001100	
EXP	0000001	001101	
LN	0000001	001110	
FAC	0000001	001111	
PIT	0000001	010000	
SIN	0000001	010001	
COS	0000001	000010	
TAN	0000001	000011	
CIRA	0000001	000100	
SINH	0000001	000101	
COSH	0000001	000110	
TANH	0000001	000111	
CIRO	0000001	001000	

ASIN	0000001	001001
ACOS	0000001	001010
ATAN	0000001	001011
CIRMA	0000001	001100
ASINH	0000001	001101
ACOSH	0000001	001110
AIANH	0000001	001111

Coroutine Jump Instruction

CRJ	0000000	00000001	CLRNO
ISTEP SUB, LABEL	00000000	0 cord+1	off set
HALT	0000000	000000	00000000

MAX	11001	
MIN	11010	
MOV	11011	
CMP	11101	

Branch Instructions

	OP	OPN
BR	00000	1000
BN	00000	1001
BNE	00000	1010
BEQ	00000	1011
BGE	00000	1100
BLT	00000	1101
BGT	00000	1110
BLE	00000	1111

Double Operand Instructions

	OP	1st OPN	2nd OPN
LT	00001		
LE	00010		
EQ	00011		
GE	00100		
GT	00101		
NE	00110		
OR	00111		
AND	01000		
NOR	01001		
NAND	01010		
XOR	01011		
EQUIV	01100		
ADD	01101		
MUL	01110		
SUB	01111		
RSUB	10000		
DIV	10001		
RDIV	10010		
RES	10011		
RRES	10100		
PWR	10101		
LOG	10110		
CIR	10111		
COM	11000		
MAX	11001		
MIW	11010		
MOV	11011		
COM	11100		
MLT	00000	00000000	00000000

Table. 2

Primitive scalar operators.

Monadic form fB			f	Dyadic form AfB	
Definition or example		*Name*		*Name*	*Definition or example*
$+B\longleftrightarrow 0+B$		plus	$+$	Plus	$2+3.2\longleftrightarrow 5.2$
$-B\longleftrightarrow 0-B$		Negative	$-$	Minus	$2-3.2\longleftrightarrow -1.2$
$\times B\longleftrightarrow (B>0)-(B<0)$		Signum	$\times$	Times	$2\times 3.2\longleftrightarrow 6.4$
$\div B\longleftrightarrow 1\div B$		Reciprocal	$\div$	Divide	$2\div 3.2\longleftrightarrow 0.625$
B	$\lceil B$ $\lfloor B$	Ceiling	$\lceil$	Maximum	$3\lceil 7\longleftrightarrow 7$
3.14	4 3	Floor	$\lfloor$	Minimum	$3\lfloor 7\longleftrightarrow 3$
-3.14	-3 -4				
$*B\longleftrightarrow (2.71828\cdots)*B$		Exponential	$*$	Power	$2*3\longleftrightarrow 8$
$\circledast *N\longleftrightarrow N\longleftrightarrow *\circledast N$		Natural logarithm	$\circledast$	Logarithm	$A\circledast B\longleftrightarrow \log B$ base A $A\circledast B\longleftrightarrow (\circledast B)\div \circledast A$

|-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 2)
○*B*←→*B*×3.14159…	Pi times	○	Circular	See table at left
~1←→0 ~0←→1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	
		≤	Not greater	Relations:
		=	Equal	Result is 1 if the relation
		≥	Not less	holds, 0 if it does not:
		>	Greater	3≤7←→1
		≠	Not equal	7≤3←→0

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

Table of dyadic ○ functions

(−*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*

Mixed-operators.

Name	*Sig*	*Definition orexample*†
Size	⍴A	⍴*P*←→4 ⍴*E*←→3 4 ⍴5←→⍳0
Reshape	*V*⍴*A*	Reshape*A* to dimension *V* 3 4⍴1⌶2←→*E* 12⍴*E*←→⌠12 0⍴E←→⌠0
Ravel	,A	,A←→(×/A)⍴A ⍴,*E*←→⌠12 ⍴.5←→1
Catenate	*V*,*V*	*P*,12←→2 3 5 7 1 2 *'T'*, *'HIS'*←→*'THIS'*
	V[*A*]	*P*[2]←→3 *P*[4 3 2 1]←→7 5 3 2
Index	*M*[*A* ; *A*]	*E*[1 3 ; 3 2 1]←→3 2 1 11 10 9
	A[*A* ; .... ; *A*]	*E*[1 ;]←→1 2 3 4 *ABCD* *E*[; 1]←→1 5 9 *'ABCDEFGHIJKL'*[*E*]←→*EFGH* *IJKL*
Index generator‡	⍳*S*	First *S* integers ⍳4←→1 2 3 4 ⍳0←→an empty vector
Index of‡	*V*⍳*A*	Least index of *A* in *U*, pr 1+⍴*V* *P*⍳3←→2 5 1 2 5 *P*⍳*E*←→3 5 4 5 4⍳4←→1 5 5 5 5
Take	*V*↑*A*	Take(drop) |*V*[*I*] first elements coordinate 2 3↑*X*←→*ABC* *EFG*
Drop	*V*↓*A*	*I*(Last if *V*[*I*]<0) -2↑*P*←→5 7
Grade-up	⍋*A*	The permutation which ⍋3 5 3 2←→4 1 3 2

Grade-down	$\downarrow A$	wouled order A(ascending or descending) $\Psi 3\ 5\ 3\ 2 \leftrightarrow 2\ 1\ 3\ 4$
Compress	V/A	$1\ 0\ 1\ 0/P \leftrightarrow 2\ 5$ $\quad 1\ 0\ 1\ 0/E \leftrightarrow \begin{matrix}1&3\\5&7\\9&11\end{matrix}$ $1\ 0\ 1/[1]E \leftrightarrow \begin{matrix}1&2&3&4\\9&10&11&12\end{matrix} \leftrightarrow 1\ 0\ 1/E$
Expand	V/A	$1\ 0\ 1/\iota 2 \leftrightarrow 1\ 0\ 2$ $\quad 10111/X \leftrightarrow \begin{matrix}ABCD\\EFGH\\IJKL\end{matrix}$
Reverse	ΦA	$\Phi X \leftrightarrow \begin{matrix}DCBA\\HGFE\\LKJI\end{matrix}$ $\quad \Phi[1]X \leftrightarrow \ominus X \leftrightarrow \begin{matrix}EFGH\\IJKL\\ABCD\end{matrix}$ $\quad \Phi P \quad 7\ 5\ 3\ 2$
Rotate	$A\Phi A$	$3\Phi P \leftrightarrow 7\ 2\ 3\ 5 \leftrightarrow {}^{-}1\phi P$ $\quad 1\ 0\ {}^{-}1\Phi X \leftrightarrow \begin{matrix}BCDA\\EFGH\\LIJK\end{matrix}$
Transpose	$V\otimes A$	Coordinate I of A decomes coordinate $V[I]$ of result $\quad 2\ 1\otimes X \leftrightarrow \begin{matrix}AEI\\BFJ\\CGK\\DHL\end{matrix}$ $\quad 1\ 1\otimes E \leftrightarrow 1\ 6\ 11$
	$\otimes A$	Transpose last two coordinates $\quad \otimes E \leftrightarrow 2\ 1\otimes E$
Membership	$A \in A$	$\rho W \in Y \leftrightarrow \rho W$ $\quad P \in \iota 4 \leftrightarrow \begin{matrix}0&1&0&1\\1&0&1&0\end{matrix}$ $E \in P \leftrightarrow 1\ 1\ 0\ 0$ $\quad \begin{matrix}0&0&0&0\end{matrix}$
Decode	$V \perp V$	$10 \perp 1\ 7\ 7\ 6 \leftrightarrow 1776$ $\quad 24\ 60\ 60 \perp 1\ 2\ 3 \leftrightarrow 3723$
Encode	$V \top S$	$24\ 60\ 60 \top 3723 \leftrightarrow 1\ 2\ 3$ $\quad 60\ 60 \top 3723 \leftrightarrow 2$
Deal‡	$S?S$	$W?Y \leftrightarrow$random deal of W elements from ιY

†Arrays used in examples:

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

1. Single operand instructions
2. Double operand instructions
3. Branch instructions.
4. Coroutine jump instructions

Ⅳ. APL to Ladder Network Translator

This Chapter program translate an APL statement in reverse polish notation to ladder network. The program can be made to simulate the instructions defined for the A-machine code, but it is abbreviated in this paper because it is a big tedious work.

Functions are divided into classes and all functions are handled by the same procedure. Only the dyadic and monadic scalar function classes contain more than one function. All other functions are handled by separate, usally small procedures.

The symbols for all the functions are defined in a file that is read when the program begins. The file also defines for each monadic and dyadic scalar function the code to be generated, the identity element, and, for the dyadic functions, whether or not the function commutes. The interpreter does not handle some of complicated monadic and dyadic scalar operators: for instance, the trigometric functions. But these can be included by adding an entry to the function definition file; Ne change of program is reqired.

Other functions not handled by the interpreter include deal, grade up, and grade down. These functions are not included because the ladder structure is not suited to their implementae tion. The program can perform the selection

operator transformations described·····.

These transformations assume that one or both of the operands are simple array references and not expressions. Abrams[1] discusses method to transform statements that do not satisfy these reqirements into statements that do. Abdrew Moulton is working on a program to transform statements like these into a format compatible with the program.

References

1. Abrams, P.S., "An APL Machine", Stanford Univ. Computer Science Department, Report STAN-cs-70-158 (1970)
2. APPLE-Ⅱ., "APPLE-Ⅱ PASCAL Programming" (1979)
3. Iverson, K.E., "A Programming Language", John Wiley & Sons, Inc (1962)
4. Jenkins, M.M., "Translating APL-An Empirical Study", Proc. of the APL Conference, Pisa Italy, June 1975, SIGPLAN Technical Committee on APL (1975).
5. Minter, C.R., "A Machine Design for Efficient Implemetation of APL", Dept. of Computer Science, Yale University, Research Report 81 (1974).
6. Perlis, A.J., "Step Toward an APL Compiler-Updated", Department of Computer Science, Yale Univ., Research Report 24 (1975).