ON THE DESIGN OF AN APL MACHINE

by

WAI KEUNG CHAN

B.S., THE CHINESE UNIVERSITY OF HONG KONG, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1975

Approved by:

Major Professor

ACKNOWLEDGEMENTS

The writer wishes to express his sincere gratitude to his major professor, Dr. Virgil E. Wallentine for his patience, encouragement and guidance during the preparation of this report and the writer's study at Kansas State University.

Page

TABLE OF CONTENTS

Page

TABLE OF ALGORITHMS

Page

TABLES

FIGURES

ABSTRACT

Microprogramming may be used to translate a high level language, e.g., APL, into an intermediate language which is then executed directly (by the microprogram). A computer microprogrammed in this way is a computer whose machine language is a high-level language. This report presents a partial design of such a machine. The high-level language chosen is APL/360. It includes a detailed description of the intermediate representation of an APL program and main routines of execution (statement scan, syntax analysis, function call and an example of the execution of an operator).

INTRODUCTION

An APL machine is a machine whose machine language is APL. It can be implemented in the hardware or by microprogramming. This report concerns a microprogrammed implementation of such a machine. Microprogramming is used to translate an APL program into an intermediate text which is then executed directly by microgrammed routines.

For the conventional compiler approach, there are usually three phases: i) syntactic analysis which checks the correctness of source statement and puts information into the symbol tables and other tables; ii) sementic analysis which disassembles the source program into its constituent parts and builds the internal form of the program; iii) code generation which translate the internal source program into assembly language or machine language. After the compilation, the code generated is then executed. Comparing to this approach, the microgrammed implementation of a high level language has the following advantages:

i) Code generation phase is eliminated and hence the translation process is simplified and speeded up.

ii) Computational thruput is improved because of the use of microgrammed exection routines.

iii) particularily for this design, the APL machine includes many checks and safeguards which the conventional compiler cannot perform.

Some quantitative and qualitative analyses of these advantages are given by Hassitt(2) and Merivin(3).

The design is quite machine independent. We assume the computer is microgrammable, with 4 bytes per word and 8 bits per byte. This report consists of 2 sections. The first

section describes the translation of APL into an internal form. The translating process resembles an assemble process. Hence we are mainly concerned with the internal representation of the program and its data. The second section concerns the execution of the translated program. We show in detail the syntactic analysis, function call and return, and an example of the execution of an operator.

This report is an interpretation, a supplementation and a variation of the design by A. Hassitt, J.W. Lageschulte, and L.E. Lyon(2). The design of the APL internal representation is based on their paper. Only a few changes have been done to the representation of a function. But the design is reorganized, filled with details and provided with many examples and ready to be adapted for implementation. The basic concept of the statement scanning routine and the use of the syntax decision table is also based on Hassitt's paper. The high-level algorithms for function call and return are designed by the author. Algorithms and flowcharts in both English and a specification language designed by the author, are presented to illustrate the concepts of the scanning routine, the function call and the function return. In order for the design to be complete, we still need memory allocation and execution routines of all APL operations. Both are machine dependent and beyond the scope of this report. However, this design provides a complete specification of an APL machine which could be easily microprogrammed.

I. TRANSLATION

The internal representation of a program is designed such that the translation is similar to a straightforward assembly process and many checks and safeguards during execution are possible. We assume there exists a scanner (a microprogram) to separate tokens of user programs. The main part of the machine is a microprogram called the supervisor/translator which translates the user program, calls functions, and operator routines, allocates memory and generates error messages.

A user's job usually has the form:

```
function definition 1
function definition 2
        .
        .
        .
function definition n
statement (which causes one or more of the
           functions to be executed)
```

Each function definition has the form:

```
∇ function header
  statement 1
  statement 2
      .
      .
      .
  statement n
∇
```

Each function header has the form:

$$R \leftarrow LA\ F\ RA\ ;\ L_1\ ;\ \ldots\ ;\ L_k$$

where R denotes the name of the result of the function (if there is no result, R and the arrow are omittes);

LA denotes the name of the left argument;

F denotes the name of the function;

RA denotes the name of the rght argument;

$L_1,\ldots,L_k$ denotes the names of the local variables.

A function may have no arguments (omit LA and RA in the header) or one argument (omit LA) or two arguments. It may have any number of local variables. Function which are defined are first translated and then saved for execution at a later time. A statement outside the scope of a function is executed immediately. It is translated as though it is the one line function

```
∇ EXEC
  statement
∇
```

The function EXEC is executed immediately.

There are two modes for the supervisor/translator, namely, the translation mode and the execution mode. The transition graph is shown in Fig. 1, where E is the execution state, T is the translation state, and the symbol on the arrow is the token encounted in scanning the user program. The arrows without any token show the transition direction upon encountering any other token except the labeled ones.

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

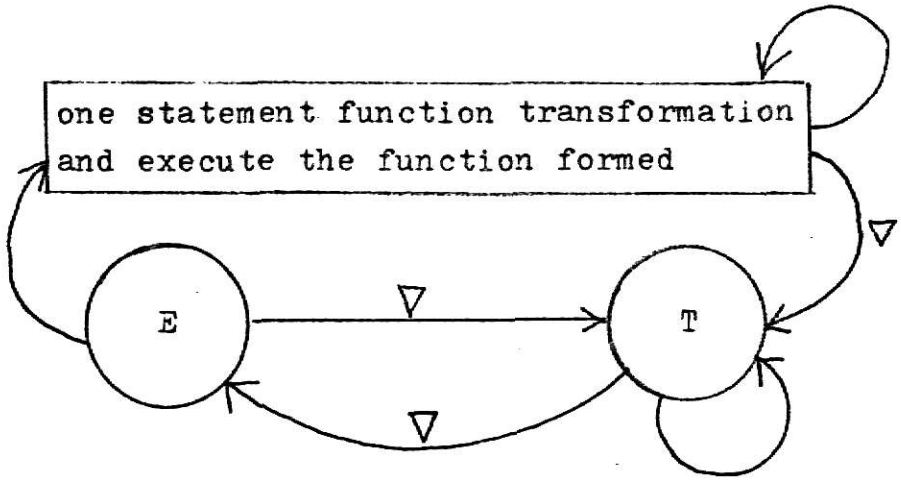


Fig 1. State Transition Graph

Let us now discuss the translation of a statement and a function. A statement may consist of variable names, function names, operators, separators, scalar literals and vector literals. We shall consider these items one by one. Items within a statement are translated into the intermediate representation in reverse order, and an 'end of statement' marker is put at the end of each statement. The marker token is the hexadecimal '0A00'. For example, the statement,

$$A \leftarrow B \times C + D$$

is translated into intermediate form as,

$$D + C \times B \leftarrow A(\text{END OF STATEMENT})$$

Once the representations of all items (variables, functions, seperators, operators and literals) are known, the translation process will be straight

1. Representation of Literal

a) A scalar literal is translated into the form:

D S

where D is a half-word descriptor which has the hexadecimal form:

7200 indicates the literal is of logical mode;
7202 indicates the literal is of integer mode;
7203 indicates the literal is of real number mode;
7208 indicates the literal is of character mode;

and S is the data which may be:

i) IBM 360 32-bit integer;
ii) IBM 360 64-bit floating point form;
iii) a character in EBCDIC form with one additional character added as padding to ensure all items begin with an even address.

By looking at the descriptor, we can decide whether it is a literal (the first 4 bits of the descriptor is 0111), and its type. Data items of different types have different descriptors. The choice of the representations of data is machine dependent.

EXAMPLE 1 Integer 26 is represented (in hexadecimal) as

7202 0000 001A

EXAMPLE 2 Real number 1.0 is represented as

7203 4110 0000 0000 0000

EXAMPLE 3 Character B is represented as

7208 C240

Note that 40 is the padding character.

b) A vector literal is represented as

D H N S

where D is a half-word descriptor such that

7220 represents a logical value;
7222 represents an integer value;
7223 represents a real value;
7228 represents character;

and H is the number of half-words used by the literal:
N is the number of elements in the vector;
S is the vector itself.

As before, the first 4 bits in the descriptor shows that it is a literal. The X'2' in th third 4-bits group shows that it is a vector. N gives the size of the vector and H helps memory allocation and deallocation.

EXAMPLE 1 Integer vector (2 4 6) is represented as:

7222 0009 0003 0000 0002 0000 0004 0000 0006

EXAMPLE 2 Real vector (1.5 1.0) is represented as :

7223 000B 0002 4118 0000 0000 0000 4110 0000 0000 0000

EXAMPLE 3 Charcater vector (N A H C) is represented as:

7228 0005 0004 D5C1 C8C3

2. All operators and separators are translated into a two-byte internal code with the following rules:

i) The first 4 bits of the internal code of an operator are,

1000 or X'8' ;

ii) The first 4 bits of the internal code of a seperator are,

1100 or X'1' ;

iii) For operator which can only be Dyadic, the 2nd 4 bits are,

0001 or X'1' ;

iV) For operator which can only be monadic, the 2nd 4 bits are,

0010 or X'2' ;

V) For operator which can be both monadic and dyadic, the 2nd 4 bits are,

0011 or X'3' .

The arrangement of these bits will greatly enhance the ease of type-checking during execution.

operator	code
+	8301
−	8302
←	8303
.	.
.	.
.	.
~	8201
.	.
.	.
.	.
∧	8101
∨	8102
∀	8103
∃	8104
<	8105
≤	8106
=	8107
≥	8108
>	8109
≠	810A

Table 1. Internal codes of operators

separator	code
(	C001
)	C002
:	:

Table 2. Internal codes of separators

3. All names are represented by a 2-byte internal name. For the Nth name in the program, its internal representation is

(Namebase + 2*N)

where we arbitrarily choose X'2000' as our name base. Hence each name(variable or function) has an internal name which is a real address of 2 bytes. We need a name table containing all names encountered in the user program in order. Whenever a name is encountered, the name table is searched. If it is not in the table, the name is added to the end of the table. Otherwise, the internal name is obtained from the table. Names in the table are separated by '*'. The internal name is a real memory location address which contains the address of the descriptor of the variable or the function.

EXAMPLE For the program beginning with

```
∇ A ← X FIT  Y; N; SUMX; SUMXX; ...
```

The name table looks likes

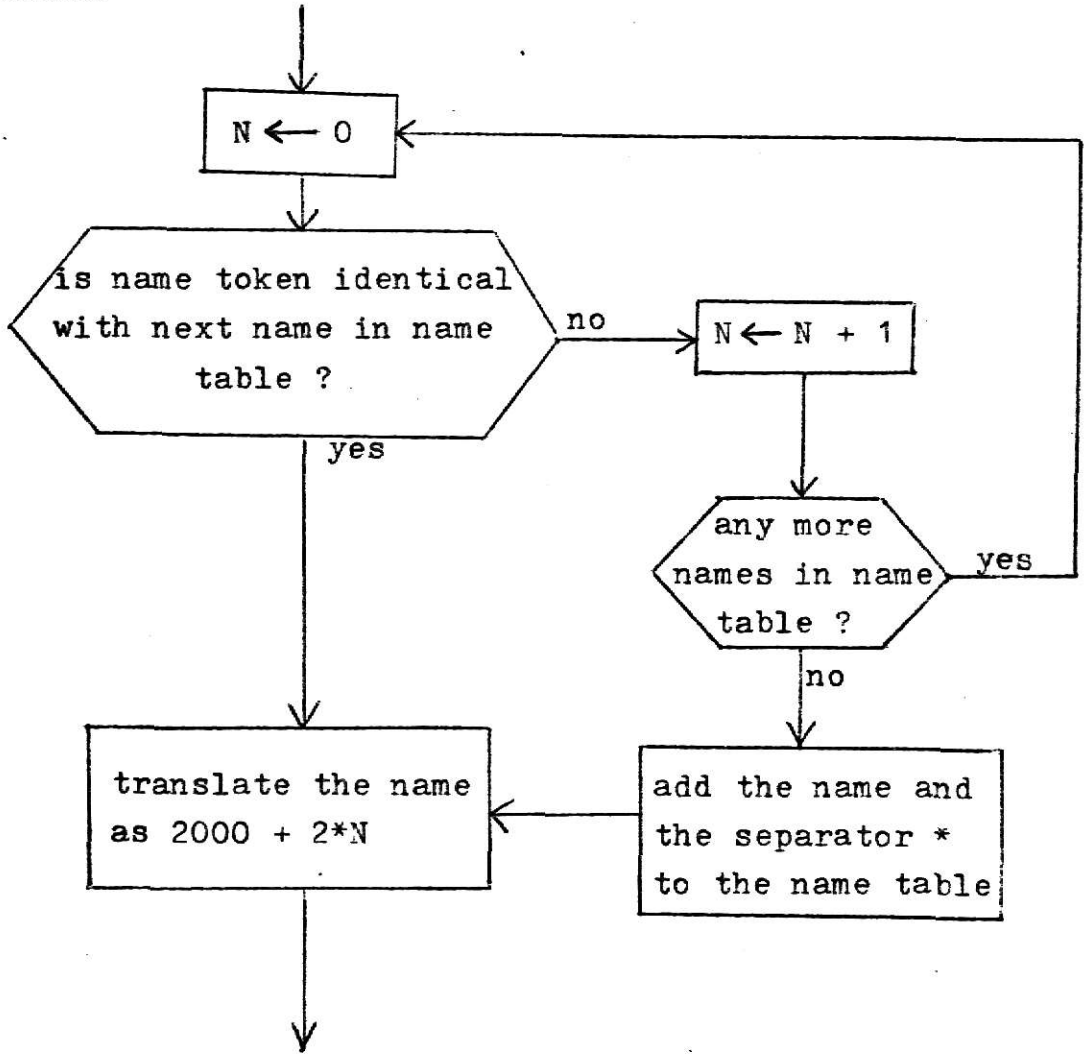
```
A * X * FIT * Y * N * SUMX * SUMXX * ...
```

and the correspondance of names and internal names is shown in Table 3.

name	internal name
A	2002
X	2004
FIT	2006
Y	2008
N	200A
SUMX	200E
⋮	⋮

Table 3. Internal names for portion of an example program.

When the supervisor/translator scans a name, it follows Algorithm 1. The search always begins with the first name in the name table.



Algorithm 1. Forming internal names for variables and function names during the scanning of statements.

4. Variables and their values

Every name corresponds to a real address, namely, its internal name. The two bytecell at this address contains the address of the descriptor of its value. A descriptor consists of 2 bytes which define the type of the value. For easy type checking, the configurations which indicate the various types are shown in Fig. 2.

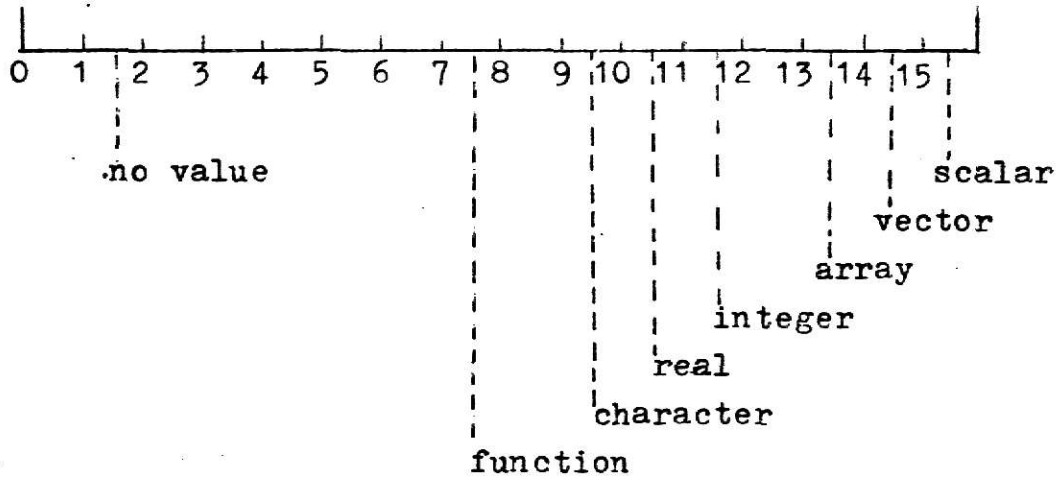


Fig. 2. Significance of bits in a descriptor of a name

Table 4 shows the different descriptors in hexdecimal.

	integer	real	character
scalar	F011	F021	F041
vector	F012	F022	F042
array	F014	F024	F043

F100 ------- function with one argument
F101 ------- function with two arguments
4000 ------- no value has been assigned to the variable

Table 4. Descriptors for variables and functions

a) Scalar

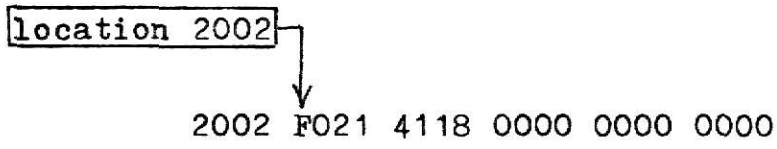
A scalar value has the form

P D S

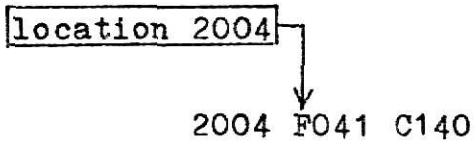
where P is the internal name of the variable;
D is the descriptor;
S may be a 32 bit integer or a 64 bit floating point representation or an EBCDIC representation of a character padded to 2 bytes (here again, the choice is machine dependent).

The internal name is the real address whose contents is the address of the descriptor.

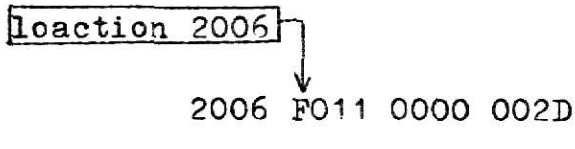
EXAMPLE 1 Real scalar 1.5 with internal name X'2002'



EXAMPLE 2 Character 'A' with internal name X'2004'



EXAMPLE 3 Integer scalar 45 with internal name X'2006'



b) Vector and Array

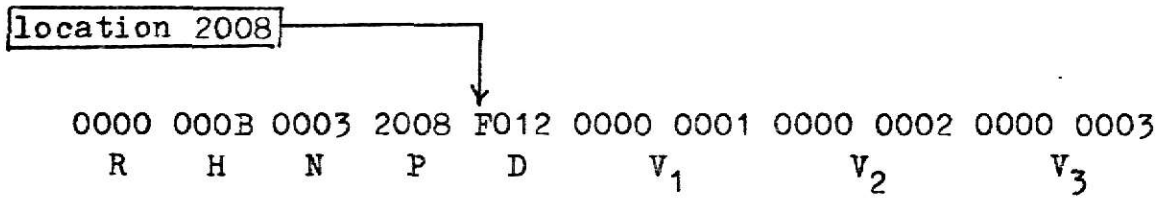
A vector or array value has the form

R H N P D V_1 V_2 V_3 . . . V_n

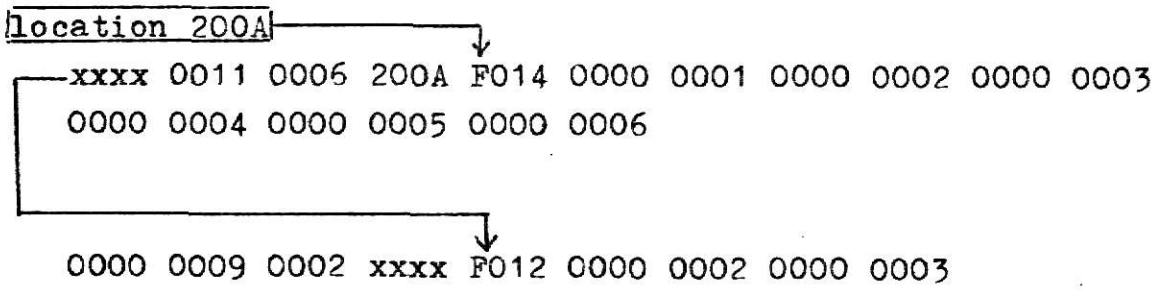
where R is the address of dimension information if item is an array, 0 if a vector;
H is the number of half-words used by R H ...V_n;
N is the number of elements in vector of array;
P is the internal name of data;
D is the descriptor.

Hence it contains the information needed for type checking, dimension checking, size checking and memory allocation.

EXAMPLE 1 Integer vector (1, 2, 3) with internal name X'2008'



EXAMPLE 2 Integer array $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ with internal name X'200A'



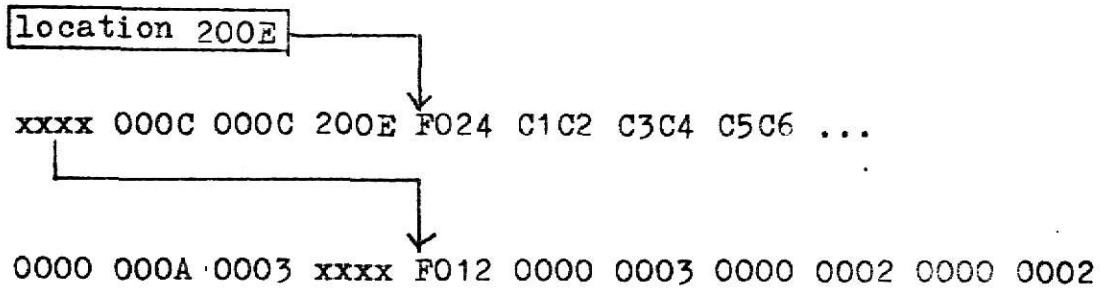
In EXAMPLE 2, location 200A contains the address of the descriptor F014. R of the value points to the descriptor of another vector, (2, 3), which shows that the array is a 2 by 3 array.

From this example, it is obvious that dimension of more than 2 is possible. This is illustrated by the next example.

EXAMPLE 3 Character array (of dimension 3 by 2 by 2)

$$\begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix}, \begin{bmatrix} G & H \\ I & J \\ K & L \end{bmatrix}$$

with internal name X'200E'



Here, R points to the descriptor of a vector (3, 2, 2) which indicates that the item with internal name 200E is a matrix.

5) Function

Since a program is represented as a set of functions in the internal representation, functions play an important role in this implementation. A function has the internal form,

$$H \quad N \quad P \quad D \quad V \quad V_1 \quad V_2 \quad \ldots \quad V_n$$

where H is the number of half-words;
N is the number of characters in the internal representation of the function;
P is the internal name of the function;
D is the descriptor X'F100', X'F101' or X'F102' depending on the number of arguments associated with the function;
V is the number of local variables in the header, which is used to speed up the process of function return;
$V_1, \ldots, V_n$ are the header , body and tail of the function.

The header of a function contains

$$S \quad L \quad R \quad L_1 \quad \ldots \quad L_n \quad O \quad B$$

where S is the internal name of the result;
L is the internal name of the left argument;
R is the internal name of the right argument;
$L_1, \ldots, L_n$ are the internal names of the local variables;
O is the half-word X'0000' to signal the end of the list;
B is the number of bytes from the descriptor, D, to the tail of the function.

If any of S, L and R does not exists, it is replaced by X'0000'.

The body of the function contains the internal representation of all statements inside the function. Following the last statement we have X'10000000' which begins the tail.

The tail contains

$$T \quad T_1 \quad \cdots \quad T_n$$

where T is the number of statements in the function;

T_1 is the offset between the end of the header and the beginning of the first statement;

.
.
.

T_n is the offset between the end of the header and the beginning of the last statement.

With the information in the tail, a branch to a statement in a function can be easily performed.

Now, we have the whoe internal representation of a function:

$$H \quad N \quad P \quad D \quad V \quad S \quad L \quad R \quad L_1 \ \ldots \ L_n \quad O \quad B \quad S_1 \ \ldots \ S_n \quad X'10000000'$$
$$T \quad T_1 \ \ldots \ T_m$$

It contains enough information to enable efficient function call and return. This will be demonstrated in the next section.

II. EXECUTION

We will discuss the syntex analysis, the function call and an example of the execution of an operator. We don't intend to present a memory mangement algorithm. APL needs a very dynamic memory management; its space requirement may change at each operation in each statement. Sample management schemes are given by Zaks(4) and Hassitt(2). Here, we assume we can be allocated the correct amount of space whenever we ask for and do not worry about garbage collection. The random access memory is divided into five sections as shown in Fig. 3.

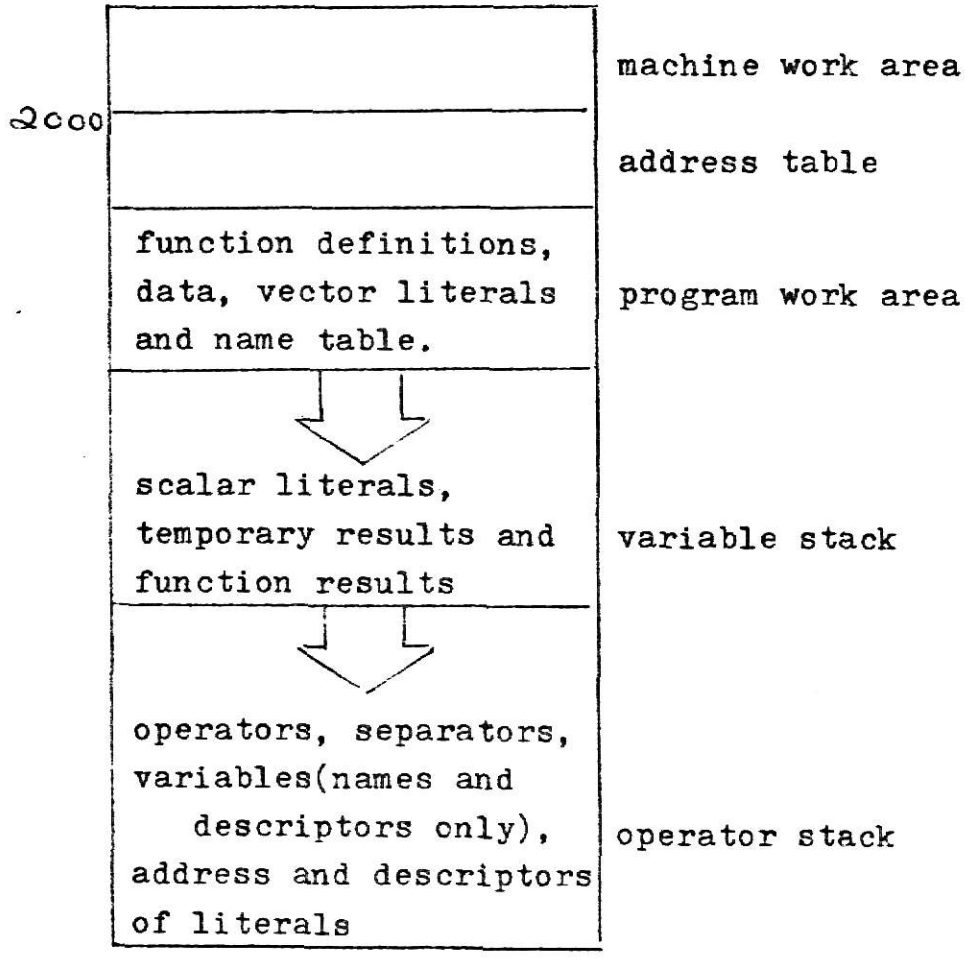


Fig.3 Division of the Random Access Memory

Use of these sections will be clear later.

1. Notation

In order to discuss the algorithms in a precise and a conceptually clear manner, we introduce some symbols and notation in this section.

The following variables are used by the supervisor/translator and reside in fixed locations in the machine work area:

NB is the address of the next two bytes of the statement being scanned;

FN is the function name of the function currently being executed;

SN is the statement number in the function FN;

BWS is the byte number in SN;

OS is the operator stack;

Vs is the variable stack.

In the following definitions, the length of each value is two bytes.

$A \Leftarrow x$ means that the value x is inserted on top of stack A;

$A \Leftarrow x,y$ is equivalent to $A \Leftarrow x$ and then $A \Leftarrow y$;

$x \Leftarrow A$ means that x is set equal to the value at the top of stack A and this value is deleted from A.

$x \overset{i}{\Leftarrow} A$ means that x is set equal to the value of the ith item from the top of stack A and all i top values are deleted;

c(a) means the content of location a (two bytes).

The pop and push operation of a stack are so useful that they are actually supported by some hardware. They can be easily simulated by microgrammed routines. Therefore we use them as basic operators in our algorithms.

For the programming-language-level flowchart, we need some additional notation.

SDT(m,n) represents the (m,n)-entry of the syntax decision table;

ω represents the current beginning address of the available block in program work area;

TYPE(x) gives the syntactical type of the item x according to Table 5a. This can be easily obtained from the first 4 bits of the item, e.g.,

for names, the first 4 bits are X'2';
for literals, the first 4 bits are X'7';
for operators, the first 4 bits are X'8' and
for separators, the first 4 bits are X'C'.

$x \xleftarrow{i} y$ means copying i half-words beginning at location y to location beginning at x.

2. Statement Scan and Syntactic Analysis

An English-like flowchart and a progrgramming-language-like flowchart of the syntactic analysis of the internal form of the user's program will be given. At the execution phase, the supervisor has the statements to be executed in internal form. It scans the statement (two bytes per token) pushes appropriate information on stacks, uses the syntax decision table to test the top two items in the operator stack to decide which action to perform. The flowchart and and the decision table are shown in Algorithm 2 and Table 5b. (These are only variations of those presented in Hassitt(2).

type	meaning
0	null, end of current part of the stack
1	operator
2	descriptor of variable
3	descriptor of function with two arguments
4	right parenthesis or bracket; another part of the separator distinguishes between these two cases
5	left parenthesis or bracket
6	;

Table 5a. Table of syntactical types(partial)

N	T=0	1	2	3	4	5	6
0	0	5	0	5	0	5	5
1	0	2	1	2	0	2	2
2	0	0	5	0	0	7	0
3	0	5	4	5	0	5	5
4	0	5	0	5	0	5	0
5	0	3	8	5	0	5	5
6	0	5	0	5	0	0	0

Table 5b. (partial) Syntax decision table. T and N are the types of the top and the next-to-top items on OS.

Actions specified in the Syntax Decision Table are:

(0) Go to scan the next item.

(1) Do a dyadic operation using the top three items on the stack.

(2) Backspace the instruction stream and the stack and do a monadic operation.

(3) Flag the fact that this is an indexed operation and continue scanning.

(4) Call a function of two arguments - the names of the function and the arguments are on the top of the stack.

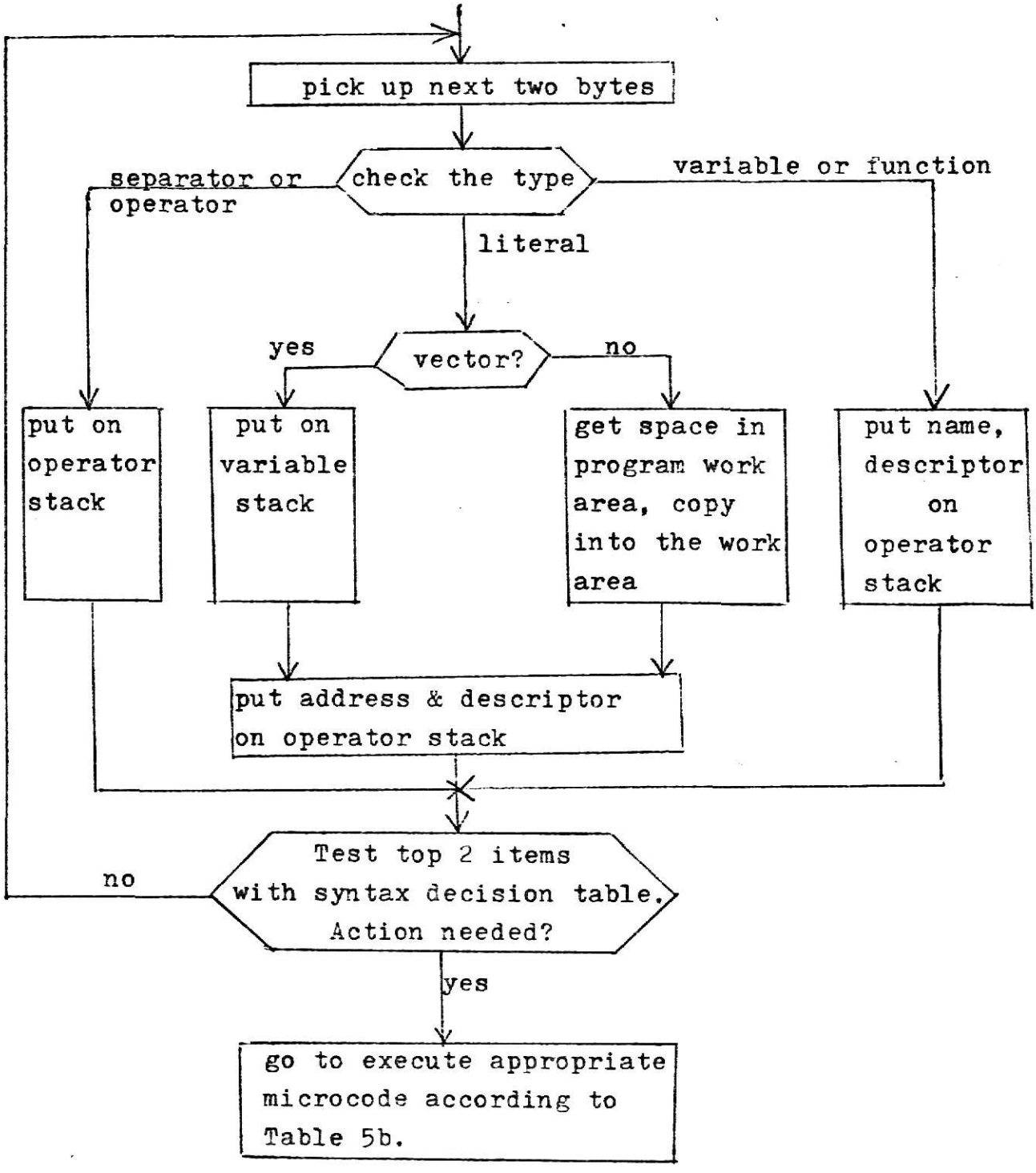
(5) Go to the syntax error microroutine.

(6) Evaluate a subscripted variable.

(7) If the top of the stack is a left bracket, go back to the scan routine, but if it is a left parenthesis, remove it and the corresponding right parenthesis from the stack, and go back to the decision routine.

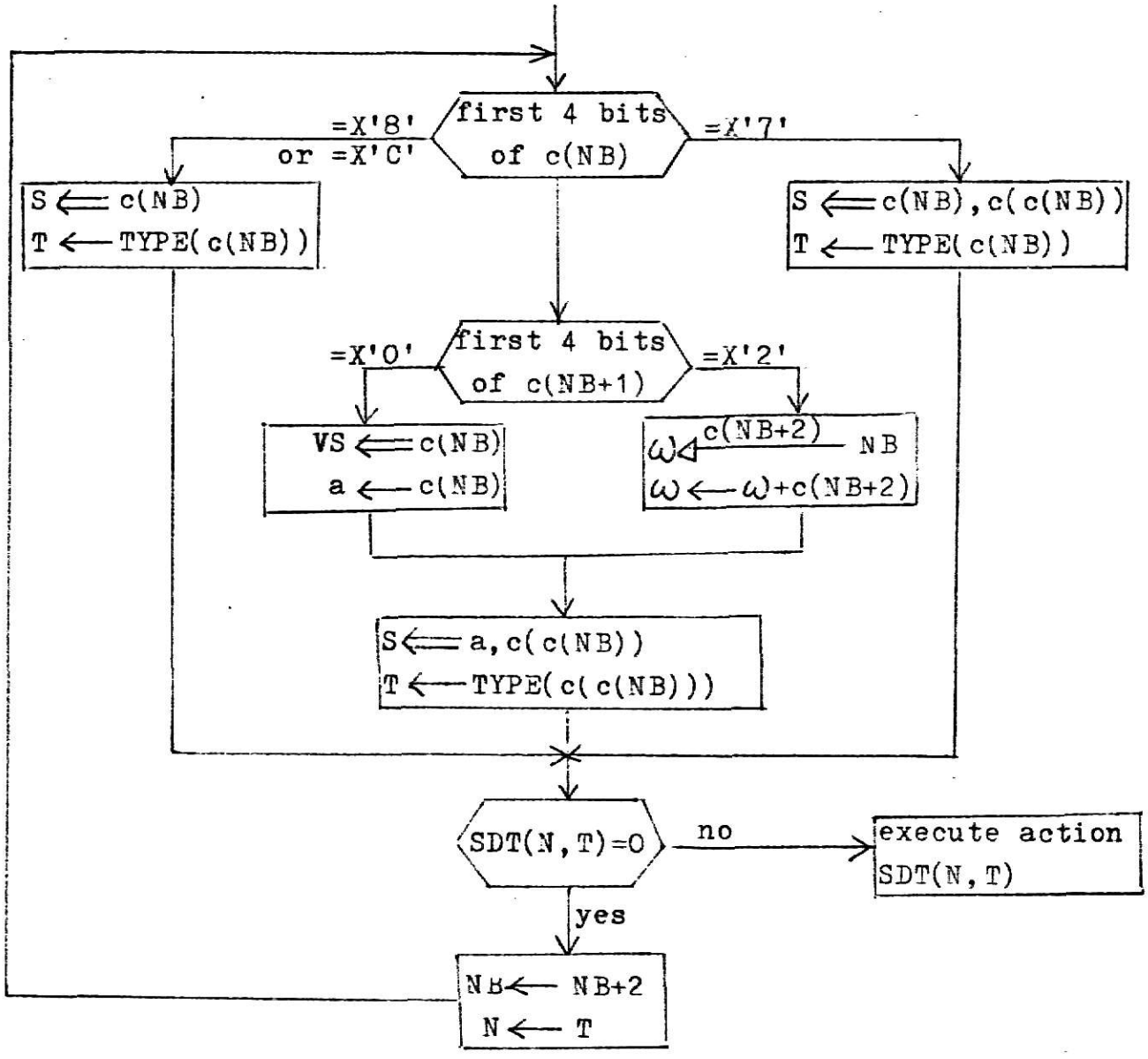
Algorithm 2. English flowchart of the scanning routine

(The machine is scanning the internal form of a statement)



Now we have the programming-language-level flowchart for the scanning routine.

Algorithm 3. Programming-language-level flowchart of the scanning routine.



ALGORITHM 3 is a refinement of ALGORITHM 2. In order to understand ALGORITHM 3, one has to be familiar with the internal representation discussed in last section and the notation introduced at the beginning of this section. The memory allocation scheme in the algorithm is simple. We assume an unlimited memory size and ignore garbage collection. Any discussion of a memory manage scheme is out of the scope of this report. The interested reader may refer to Zaks(4) and Hassitt(2).

3. Function Call and Function Return

Before we discuss the implementation of the function call and return, let us discuss the function structure of APL. All APL functions are defined at the same global level(level 0 or program level). An APL function may introduce new local variables, which are explicitly defined in its header, and reference former local variables introduced by a previously called function (lower level). A local variable thus introduced by a function FN is said to made global (and therefore accessible) to all functions called during the activation of its owning function FN. These variables are accessible to all the functions called by FN, but, when FN returns, none will be transmitted back to the calling level; their effect is local to FN. A function can also be called recursively (this can be done directly or indirectly).

In our implementation, at level 0, all variables are global by the way we translate all the names. At a function call, all old values of the function result, arguments and local variables must be saved on the operator stack. New values of arguments are passed by changing pointers in the address table. New values of function result and local variables are set to 'no value'. Old execution status (which function, which statement and which byte being executed) must also be saved for function return and this is put on the operator stack with a marker. New status is set to the first byte of the first statement of the function being called. When a function executes a return, the old status and old values of the result, arguments, and local variables are at the top of the operator stack and can therefore be restored.

In discussing the general function activation and termination concepts, let us first illustrate the function call and return of a function of one argument. At the point of function call, according to ALGORITHM 3, we have, at the top of the operator stack,

(rest of stack) t dt F dF (top of stack)

where F and dF are the internal name and the descriptor of the function respectively;
t and dt are the internal name and the descriptor of the argument repectively.

The algorithm is given in a PL/1-like language. The stack operations and assignment operations used in the algorithm can be easily replaced by microroutines. Before presenting the algorithm, let us recall the internal representation of a function. It is

H N P D V S L R L_1 ... L_n O B S_1 ... S_m X'10000000'
T T_1 ... T_m

With this in mind, one can easily understand the algorithm.

ALGORITHM 4 Function call of function with one argument

```
T <=2= S;                          Remove function name from the stack.
T1 <=2= S;                         Remove argument from the stack.
a <-- c(T)+4;                      Address of S in function header.
do i=1 to 3;                       Save old values of result, argument
   S <== c(c(a));                     and set new result and argument
   c(c(a)) <-- X'4000';               to 'no value'.
   a <-- a + 2;
end;
do while (c(a)¬=X'0000');          Save old values of local variables
   S <== c(c(a));                     and set new local variables to
   c(c(a)) <-- X'4000';               'no value'.
   a <-- a + 2;
end;
c(T) + 8 <-- T1;                   Set argument to new value.
S <== FN;                          Stor current status: function name;
S <== SN;                                statement number;
S <== BWS;                               byte number and
S <== '#';                               marker.
FN <-- T;                          Set new status: function name;
SN <-- 1;                                statement number;
BWS <--0;                                byte number.
NB <-- a + 4;                      Next byte to be scanned.
(return to scanning routine)
```

ALGORITHM 5 Return from a function with one argument

```
t ⟵ c(c(FN) + 4);                    Push internal name and
S ⟸ t, c(t);                         descriptor of result on stack.
t ⟸ S;
if t¬='#' then error;
BWS ⟸ S;                             Restore old status;
SN ⟸ S;
FN ⟸ S;
a ⟵ c(FN) + c(c(FN) + 2)*2 + 8;      Address of last local variable
                                     in the header.
a1 ⟵ a + 4;                          End of header.
do i=1 to c(c(FN) + 2);              Restore old result, argument
   c(a) ⟸ S;                         and local variables.
   a ⟵ a-2;
end;
NB ⟵ (a1 + 2) + c(c(FN) + c(a1) + 2*(SN + 1)) +BWS
                                     NB is the next byte to be
                                     scanned. (a1 + 2) is the end
                                     of header. The other part at
                                     the left hand side is the
                                     offset of the statement SN
                                     from the end of the header.
```

The procedure for the call of a function with 2 arguments is almost the same and is given as ALGORITHM 6. Function return is the same as ALGORITHM 5. In this case, at the time of function call, at the top of the operator stack, we have,

$$t_1 \quad dt_1 \quad F \quad dF \quad t_2 \quad dt_2 \quad \text{(top of stack)}.$$

Because of the simplicity of operations involved in these algorithms, the corresponding microroutine will be very fast and efficient. In the algorithm of function return, we ignore the deallocation of space used by the function. This can be done by a memory management routine.

ALGORITHM 6 Fuction Call of funtion with two arguments

```
T2 <<=2= S;
T  <<=2= S;
T1 <<=2= S;
a <-- c(T) + 4;
do i=1 to 3;
   S <<= c(c(a));
   c(c(a)) <-- X'4000';
   a <-- a + 2;
end;
do while (c(a)¬=X'0000');
   S <<= c(c(a));
   c(c(a)) <-- X'4000'
   a <-- a + 2;
end;
c(T) + 8 <-- T2;
c(T) + 6) <-- T1;
```

```
S ⟸ FN;
S ⟸ SN;
S ⟸ BWS;
S ⟸ '#';
FN ⟵ T;
SN ⟵ 1;
BWS ⟵ 0;
NB ⟵ a + 4;
(return to scanning routine)
```

4. Execution of a Simple Operator

There are many operators in APL. The design of execution routines for all these operators is out of the scope of this report. However, we give an example of the execution of the operator, +, which is given by Hassitt(2).

Suppose we are at the stage where the operator stack is

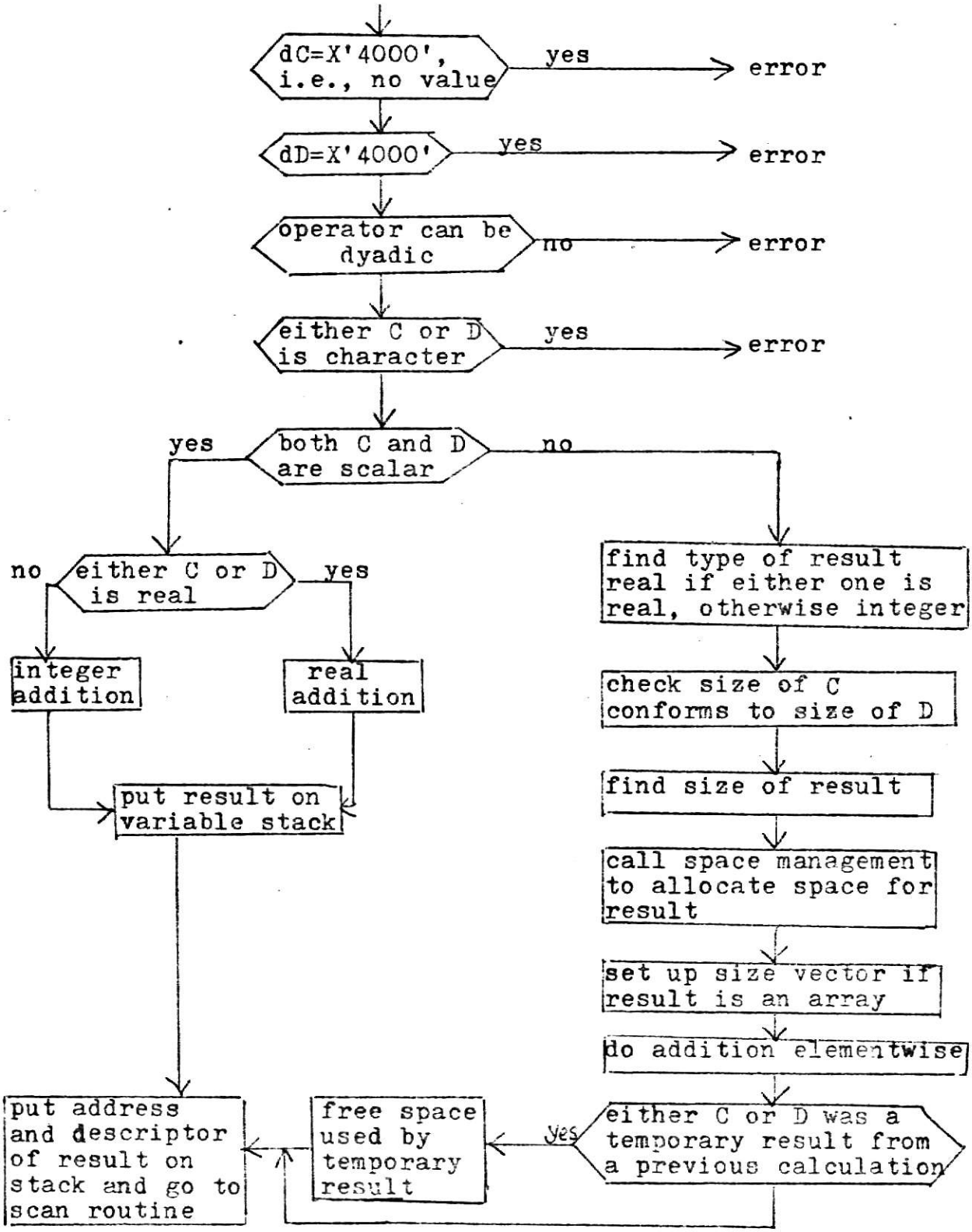
null D dD + C dC (top of stack)

where D is the two-byte internal name for D;
C is the two-byte internal name for C;
dD is the two-byte descriptor for D;
dC is the two-byte descriptor for C.

The syntax analysis indicates that action(1) is required: that is, execute a dyadic operation. The execution procedure is shown in ALGORITHM 7. In the flowchart, we have to check that the size of C conforms to size of D. The size rules are: a scalar can be added to a scalar, a vector, or an array; a one-element vector of one-element array obeys the same rule as a scalar; vector plus array is not allowed; vector plus vector is allowed if both have the same number of elements; array plus array is allowed if each of their dimensions matches.

Even from the execution of such a simple operator, we can see the time spent on type checking and apace management. However, with the design discussed in this report, writing microcode routines for these operators will not be difficult because all necessary informations are well organized in the internal representation of the program.

ALGORITHM 7 Execution of a simple operator



CONCLUSIONS

Since a complete design of the APL machine is beyond the scope of this report, we do not look into the implementation of each operator in APL. Writing these routines will be a lengthy but not difficult job. With the design of this report, the dynamic checking of types, size of vectors and arrays, validity of operators and validity of function calls is entirely possible. We have not investigated memory management carefully, but we note that implementation of this machine in a virtual memory environment is feasible.

A virtual memory system can be implemented within the APL machine. The machine will execute in virtual locations. The set of these locations is called the address space. The set of real memory locations is called the memory space. What we need is an address map, f, from the address space to the memory space as indicated in Fig. 4.

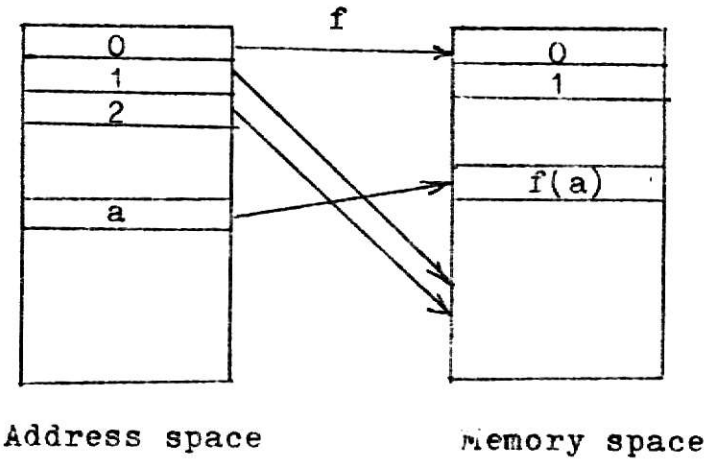


Fig. 4 The address map

For example, in a paging environment, f will be a page map. At each address reference, the map is invoked to give the real address. We notice that in the algorithms of our machine, calculations involving addresses are often performed in order to obtain information from a particular part of a data item. But these do not invoke the address map. This is illustrated in the following statements of ALGORITHM 4.

The right hand side of the statement,

$$a \leftarrow c(T)+4$$

calculates an address. The result, c(T)+4, will be regarded as a value and stored to f(a), the corresponding location in the memory space. We notice that the address is not invoked to change c(T)+4 to an address in the memory space because it is a value instead of a reference to a location. But we have to invoke f to acquire the real address, f(a), of a, and store the result c(t)+4 into it. Later in the statement,

$$S \Leftarrow c(c(a))$$

We have to invoke f to obtain the contents of f(a) and then the content of f(f(a)) in order to obtain the content of the location in the right hand side. This example clarifies the distinction between an address as a value and an address as a location. This difference is critical when the system is implemented within a virtual memory environment.

Section I of this report is in detail and ready for use to build such a machine. Concepts in Section II are presented clearly in a language which can be easily converted to microcode. It is hoped that this report can serve as the basis for the actual microprogramming of an APL machine.

REFERENCES

1. Gries, D. Compiler Construction for Digital Computer. Wiley, 1971.
2. Hassitt,A., Lageschulte,J.W., and Lyon,L.E., Implementation of a High Level Language Machine. CACM 4 (April 1973), pp. 199-212.
3. Merwin,R.E. Direct Microprogrammed Exection of the Intermediate Text from a high-level language compiler. SIGPLAN NOTICES Vol 9, 8(Aug. 1974), pp.145-153.
4. Zaks,R. Dynamic Memory Management for APL-like Languages. SIGPLAN NOTICES Vol 9, 8(Aug. 1974), pp.130-138.
5. Knuth,D.E. The Art of Computiner Programming. Vol 1. 2nd edition.Addison Wesley, 1973.
6. Rosin, R.F. Contemporary Concepts of Microprogramming and Emulation. Computer Surveys 1, 4(Dec. 1969), pp. 197-212.

ON THE DESIGN OF AN APL MACHINE

by

WAI KEUNG CHAN

B.S., THE CHINESE UNIVERSITY OF HONG KONG, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1975

ABSTRACT

Microprogramming may be used to translate a high level language, e.g., APL, into an intermediate language which is then executed directly (by the microprogram). A computer microprogrammed in this way is a computer whose machine language is a high-level language. This report presents a partial design of such a machine. The high-level language chosen is APL/360. It includes a detailed description of the intermediate representation of an APL program and main routines of execution (statement scan, syntax analysis, function call and an example of the execution of an operator).