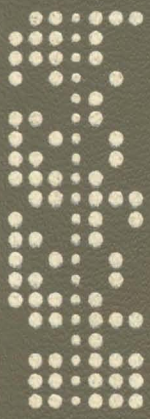


NELIAC
PROGRAMMING GUIDE

September 1962 ● Author: A. E. Lemay, LTJG USNR ● Editors: H. C. Kerr, Jr., A. M. Osborne, LT USN

U. S. NAVY ELECTRONICS LABORATORY, SAN DIEGO 52, CALIFORNIA

FOREWORD

This document describes NELIAC (Navy Electronics Laboratory Algorithmic Compiler) as developed on the UNIVAC AN/USQ-17 computer and applied on the UNIVAC AN/USQ-20 and UNIVAC 490 real time computers.

The author is indebted to LT J. E. White, USN, LCDR R. R. McArthur, USN, LT K. S. Masterson, USN, and Mr. Roger Rempel, who were instrumental in the development of the NELIAC language and compiler. The NELIAC language itself was invented and developed in its early phases by Dr. H. D. Huskey, Dr. M. H. Halstead, and LCDR R. R. McArthur, without whom there would be no NELIAC language.

Special acknowledgment is due Mrs. Helen Bate for the preparation of the manuscript and her knowledgeable editing of the text for technical accuracy.

INTRODUCTION

The availability of large scale automatic data processing equipment has revolutionized problem-solving techniques in virtually every modern industry and laboratory.

Programming is the operation by which such data processing equipment is instructed to perform a particular task or sequence of tasks. Internal operation is in a numerical code or -machine language- . Usually, no distinction is made between the data which is operated on and the instructions themselves. Instructing the machine in its own language can be an onerous and monotonous job. To simplify the programming task, many -automatic programming- systems have been developed. These -systems- are programs written for a computer to simplify the problem of programming for it or other computers. The term -compiler- is used to refer to such a system, and particularly to one which accepts an English or algorithmic input language. The compiler then translates that higher level language to the basic language of a particular computer.

The evolution of programming systems has progressed further and further away from the characteristics of the machines themselves. Indeed, some languages have highly machine independent characteristics, and may successfully generate programs for several dissimilar computers. NELIAC is a dialect of the ALGOL 1958 language; and is classified as a procedure-oriented language.

TABLE OF CONTENTS

TABLE OF CONTENTS

I. NELIAC SYNTACTICS

A. SYMBOLS

NELIAC programs are written using a basic set of alpha-numeric, arithmetic, and punctuation symbols. The symbol set consists of 26 letters, 10 numbers and 25 arithmetic-punctuation symbols.

These symbols are translated from the code of the input device (Flexowriter, Teletypewriter, or Hollerith card reader) into compiler code. This code table is a logically arranged table which represents each symbol with the 6 bits necessary in the binary mode of the computer. Table I lists all the NELIAC symbols with the NELIAC internal codes. When punched cards are used for preparing NELIAC programs, special composite symbols are required. These are also illustrated in Table I. Simple definitions of each NELIAC symbol are given in Appendix A; more definitive explanations are given in the following sections.

B. GRAMMAR

The compiler, itself a sophisticated program written in its own language, is classified as a -self compiler-.[1] It utilizes -current operator/operand/next operator- combinations to transform the procedure oriented language into the computer oriented object program; see Table II. It is therefore absolutely necessary for the programmer to observe rather strict rules of punctuation.

1. The use of the , - , is as a quotation mark and is only for the convenience of the reader.

The rules are, however, simple and quite consistent throughout the framework of the language. Misuse of operators, i.e. punctuation, in the NELIAC language will result in more serious implications than merely using bad grammar, and will cause diagnostic printouts at compiling time. Indeed, an error in punctuation may cause a great many messages describing syntactical errors which are caused directly by the first error in the chain.

NELIAC programs consist of up to three elements: the Declarative Statement, the Dimensioning Statement, and the Flowchart Logic.

1) The Declarative Statement is a means of putting machine dependent operations into NELIAC language without using machine language in the flowchart logic.

2) The Dimensioning Statement or noun list contains the assigned names(nouns) of all constants, variables, lists, and tables, etc. used in the flowchart logic.

3) The Flowchart Logic is the NELIAC operating program itself. The flowchart logic is written using NELIAC symbols, constants, predefined variables (nouns), and other routine and subroutine names (verbs). Usually, programs are of such extent that they will consist of a collection of flowcharts (flowchart logic) with their associated dimensioning statements along with one declarative statement from which the compiler manufactures a machine coded program.

C. NAMED VARIABLES OR NOUNS

Four types of variables are commonly used in NELIAC programs. They are: (1) signed whole words, (2) unsigned half word or bit fields, (3) multiword floating point quantities, and (4) address variables. Half words and bit fields are always treated as positive integers. Constants in legal number format may be used in any part of the flow-chart. See the Section on dimensioning for details.

D. NAME FORMAT

The first fifteen characters in any NELIAC name are significant, not including spaces. Any character past the fifteenth is disregarded. Names must begin with a letter; thereafter, any combination of letters or numbers constitutes a legal name. The single letters -i- through -n- are register variables B1 through B6, and therefore cannot be used as labels or -verbs-. Operators cannot be used as names, i.e. those in the NELIAC card symbol set: BEGIN, END, OCT, etc.

Examples:

LEGAL NAMES	ILLEGAL NAMES
q|zt	1 q|zt
z 999876 b	1
an extra long legal name	cde]pql

There are three levels of name precedence in the NELIAC language:

1) Permanent or global names
2) Flowchart local names
3) Formal parameter and subroutine names

A global or permanent name is one that has been defined in the dimensioning statement or is the name of a procedure and may be referenced anywhere within the system program. Names that are local to a flowchart contain a temporary sign -|- and may not be referenced outside the flowchart.

If a name is defined in a function or subroutine, it may not be referenced outside that function or subroutine.

When the programmer uses the same name in two or three levels of name precedence, the compiler uses the definition of the most local name.

E. NUMBER FORMAT

Numbers may be used as operands in the flowchart logic with the restriction that no floating point numbers with exponent parts are allowed. See example 1. The dimensioning statement has no such restrictions. Example 2 illustrates all the legal forms.

Example 1: IN FLOWCHART LOGIC

LEGAL NUMBERS	ILLEGAL NUMBERS
3777777777_8	123451234512_8 (more than 10 octal digits)
7777_8	
12345	
3.2	.321 (no leading number or zero)
99.99	3.731_8 (no floating point octal
0.00000123	are allowed)

Example 2: IN THE DIMENSIONING STATEMENT

LEGAL NUMBERS (include all above legal numbers)

3.0×74 (a decimal power of ten is understood)

3×-1

4×0

-33_8

- 89

-62.3×-3

- 0 (octal 7777777777)

F. PUNCTUATION

A program written in the NELIAC language depends upon the proper use of punctuation. The -punctuation symbols- used are shown below.

,	comma	}	right brace
;	semicolon	{	left brace
.	period	∩	boolean -and-
:	colon	∪	boolean -or-

1) COMMA: Commas are universal separators, they are used to show the end of a phrase or sequence. Commas can be used with great freedom almost everywhere.

Example 1:

, Compute Tax, Sum, Z,

In Example 1 the comma indicates a transfer operation. Here -compute tax- , -sum-, and -z- have been defined as subroutines. The example says -transfer to compute tax and come back-, etc. Note that the return transfer is implied only when the previous operator is -punctuation-.

Example 2:

A + B → C → D,

Example 2 shows that punctuation usually follows the final operand in all store operations.

2) SEMICOLON: The semicolon denotes the end of the dimensioning statement. The compiler considers everything following the semicolon at the end of the -dimensioning statement- to be flowchart logic.

The semicolon also indicates the end of a true or false alternative to a comparison as described in Section III-B.

The other legitimate use of the semicolon is to separate the input parameters from the output parameters in a -function call- or a -function definition-. See Section III-E and F.

3) PERIOD: The period indicates an unconditional transfer when the previous operator is -punctuation-.

Example 1:

, Procedure.

Like the semicolon, the period indicates the end of a true or false alternative wherever it is used. A misplaced period, i.e. one not indicating an unconditional transfer, will terminate an alternative as effectively as a semicolon. See Section III-B.

A double period signifies the end of a flowchart and will generate an unconditional jump stop to the flowchart entrance.

4) COLON: The colon indicates definition when the previous operator is -punctuation-.

Example 1:

; Compute Number: --------

The example defines that which follows the colon as the subroutine or routine associated with -compute number-.

When the colon is preceded by one of the comparison operators (=, >, etc.), it indicates the beginning of the -true-alternative. See Section III-B.

In the dimensioning statement the colon has several other defining capabilities as shown in Section II-B, E, F, and H.

5) LEFT AND RIGHT BRACES: The left and right braces are symbols which indicate loops (Section III-C) or subroutines (Section III-E). In all other respects they are identical to commas. For their application in dimensioning statements see Section II-B, C, D, E, and G.

6) BOOLEAN AND-OR: The boolean operators are symbols which separate parts of a compound decision. In this sense they are treated as punctuation. See Section III-B.

II DIMENSIONING

A. DIMENSIONING STATEMENT

The dimensioning statement, often called the noun list, contains the assigned names (nouns) of all variables, constants, lists, and tables. Dimensioning is the process of allocating machine locations and naming variables; stating whether or not they initially have known numerical values and their mode, i.e. fixed or floating point, and information on the forms and lengths of any lists or arrays. All variables used in a NELIAC program must be defined at some point or other; however partial words and floating point variables must be defined before they are used. The dimensioning statement may be omitted in certain cases, but in any event a semicolon must precede the flowchart logic.

B. SINGLE ITEMS, LISTS, TABLES, CONSTANTS, AND PARTIAL WORDS

Example 1:

```
a, b. c.
```

This example defines the fixed point full word variable -a- and the floating point variables -b- and -c-, all equal to zero.

Example 2:

```
a(20), b(20).
```

In example 2 -a- is a list of 20 fixed point full word variables, -b- is a list of 20 floating point variables. In both cases all locations are equal to zero.

Example 3:

a(10) = 1, 7, 3, 6,

Example 3 shows the general technique by which a list may be wholly or partially allocated with non-zero quantities. The remaining six locations of list -a- are equal to zero.

Example 4:

Z(10) = 2.1, 3.2, 0.15, 100 × -10,

The mode of a list, i.e. fixed or floating point, is determined by the first numerical value assigned in that list. Example 4 illustrates a list -Z- of 10 floating point items, four of which are assigned non-zero values. (Example 3 illustrates a list of 10 fixed point items.)

Example 5:

A: B: {c(24→29), d(24→29), e(0→17), f(12→17),} (100),

In example 5, the partial word variable -f- occupies bits 12 through 17 of the computer word -B- which is also defined as -A-. Note that -f- is wholly contained in the variable -e-, that -c- and -d- occupy the same bit locations and that bits 18 through 23 are unallocated. Note also that there are one hundred of each of these variables, all of which are numerically equal to zero.

C. JUMP TABLES

Example 1:

Jump Table = { P, Q, R, S, T, U, V, },

In the example Jump Table [0] contains the address of -P-,

successive entries contain the addresses of the routines -Q- through -V-. The names mentioned in the jump table can be names of routines or subroutines. In the flowchart logic statement, Jump Table[i], generates a return jump to the address specified in the lower half of the ith entry in the dimensioned jump table. Jump tables may be used to execute subscripted return jumps or straight jumps depending upon the items defined.

D. ADDRESS SWITCHES

Example 1:

switch = {a},

Address switches and jump tables are identical in principle. Switch[0] contains the address of the noun -a- and the k-designator (Note 1) appropriate to the noun. The functions of jump tables and address switches should not normally be mixed.

Example 2:

Switch = {Noun 1, Noun 2, Noun 3, ...},

The switch[0] contains the address of Noun 1. Switch[1] contains the address of Noun 2, etc. Both jump tables and address switches are address variables, i.e. an address rather than data is referenced. The distinction between the two exists only in their usage. A Jump Table may be used as an address switch to obtain the address of a routine, but using

Note 1: The k-designator, inserted in bits 18 through 20, is for operand interpretation in an AN/USQ-20 machine instruction.

an address switch as a jump table is very inadvisable, since one rarely wishes to jump into the dimensioning statement area.

E. CONGRUENT TABLES AND LISTS

Example 1:

```
A(10): B(5), C(5),
```

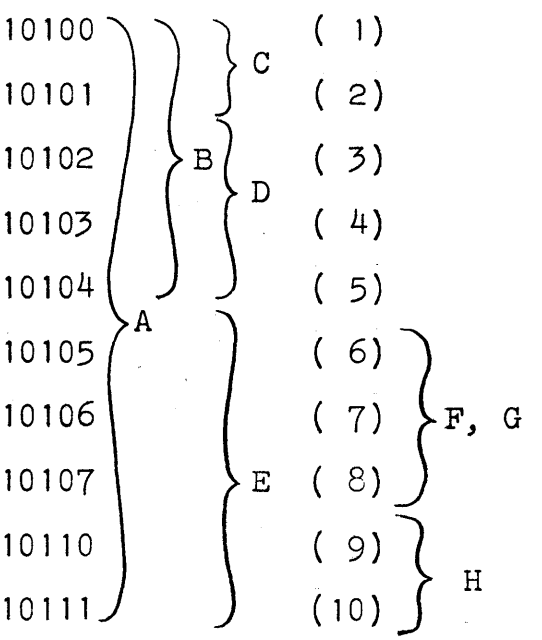
In example 1 the lists -B- and -C- are both numerically equal to zero and are wholly contained in the list -A-. The element -A[5]- is the same as element -C[0]-. The programmer must ensure the allocation of the whole of list -A- with other lists or tables, as in example 2.

Example 2:

```
A(10): B(5): C(2) = 1, 2, D(3) = 3, 4, 5, E(5):
{F(24→29), G(0→5),} (3) = 6, 7, 8, H(2) = 9, 10,
```

Example 2 illustrates some of the power of this technique. Note that only ten cells have been allocated in the object program. Example 3 illustrates diagrammatically the relations expressed in example 2.

Example 3:



F. DOUBLE INDEXING AND TWO DIMENSIONAL ARRAYS

Example 1:

A (3 × 4): A0(4) = 0, 1, 2, 3,
A1(4) = 4, 5, 6, 7,
A2(4) = 8, 9, 10, 11,

Example 1 illustrates a two-dimensional array of three rows and four columns. Each row is named with the appropriate row number. Thus, -A0- is row zero of the array -A-. Note that the leading element of the array is -A[0,0]-. Elements of two-dimensional arrays are stored as they appear on the flowchart, i.e., arrays are stored sequentially by rows.

Example 2 illustrates some of the identities possible with the conventions adopted. Note that the last element of a two-dimensional array has subscripts which are one less than the number of rows and columns dimensioned.

Example 2:

A[0,0] is the same as A0[0] and is equal to 0
A[2,1] is the same as A2[1] and is equal to 9
A[2,3] is the same as A2[3] and is equal to 11

G. TWO DIMENSIONAL JUMP TABLES

Example 1:

Q(3×3) = {A, B, C,
D, E, F,
G, H, P,}

Table -Q- is like any other jump table except that it may be referenced in the fashion of example 2.

Example 2:

```
, go to Q[1, 2] .
```

This example will result in a transfer to routine -F-.

H. LITERALS

Literals are defined in the dimensioning statement and stored in memory as NELIAC code just as they are written. They are useful for alpha-numeric headings and output formats, etc. The data contained in the literal begins with the first character after the colon and ends with the character just before the right bracket. Any NELIAC symbol may form part of the literal except the right bracket, which ends the literal.

Example 1:

```
[Text: This is a line of text, a+b→c,]
```

The name -text- is an address variable. Whenever -text- is used as an unsubscripted noun, the address of the literal will be obtained. The literal is formed internally as NELIAC code, packed five characters to a word from left to right. A full zero cell follows the literal.

III. FLOWCHART LOGIC

A. COMPUTATION RULES

NELIAC is an algebraic language; the rules of arithmetic precedence are strictly observed. The order of execution within an algebraic group is:

1. Scaling (× 2↑ or / 2↑)
2. Multiplication or division (× or /)
3. Addition or subtraction (+ or -)

Example 1:

A × B + C → P

The above example says: Take the product of A and B, add C, then store in P.

Example 2:

A / B / C → P,

Example 2 says: Divide A by B, then divide the resultant quotient by C, then store in P.

Example 3:

A - B × C + D → P,

Example 3 says: Subtract from A the product of B and C, then add D and store in P.

A series or combination of divides and multiplies is taken from left to right.

Example 4:

A / B × D / C → P,

This example is interpreted as A divided by B, multiplied by D, divided by C, stored in P.

The programmer should observe that A, B, C, and D may be expressions enclosed in parentheses.

Example 5:

$$(6+H) \times (F+K) + (6-F)$$

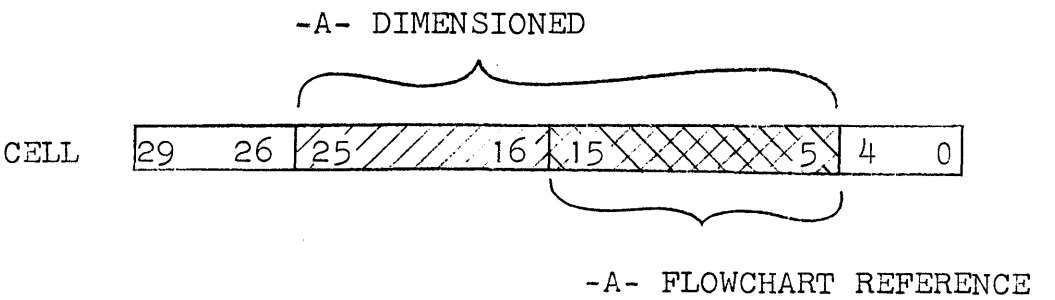
Example 5 is treated like example 1 after the grouping is evaluated. However, an expression enclosed in parentheses must contain at least one arithmetic operator.

Note that -H- and -F- may also be expressions, but -K- is the register variable B3. Any single variable may have been previously defined as a bit field, a whole or half word, an address variable, or a floating point quantity. Mixed, i.e. fixed point and floating point, operations are not permitted.

The programmer may refer to specific bit locations on any fixed point word.

Example 6:

DIMENSIONING	FLOWCHART
CELL: { A(5→25),},	A(0→10) + ...



Any fixed point expression may be shifted or scaled by multiplying or dividing that expression by a power of 2.

Example 7:

```
A(3→14) × 2↑6 → P,
A × 2↑I → P,
B / 2↑A → P,
```

Example 7 shows the three legal shifting operands: constants, register variables, or whole or half words.

The programmer should observe that the CO/NO combinations of (× 2↑ or / 2↑) have the highest arithmetic precedence and are always executed first.

The result of every computation must be stored in a variable by the use of the store operator.

Example 8:

```
A + B → C,
```

Expressions in a decision statement (see Sec III-B) need not be stored in this fashion.

An expression may include a store operator at any point.

Example 9:

```
(A + B → C) / (D + E → F) → Q,
```

Algorithms of dissimilar mode, i.e. fixed or floating, may be separated by any punctuation or the right arrow. When a fixed point expression is stored in a floating point variable, the normalized floating point representation of that integer is obtained. When a floating point expression is stored in a fixed point variable, the truncated integer value is obtained.

B. DECISION MAKING

There are seven basic decisions for expressions of similar mode:

A = B:

A ≠ B:

A < B:

A > B:

A ≦ B:

A ≧ B:

A < B < C: (fixed point only)

Note that A, B, and C can be symbolic expressions of the type discussed in Section A.

Example 1:

```
A = B: (true alternative - any expression);
       (false alternative - any expression);
```

The true or false alternatives may be terminated by a semi-colon or by a period. The expression in the -true- or -false- alternative can be another decision, if desired. For clarity it is permissible to enclose the whole true or false alternative in a set of braces.

Example 2:

```
A = B: { True };
       { False };
```

Obviously, in executing a branch statement of this type, either the -true- alternative or the -false- alternative will be executed, but under no circumstances will both ever be executed.

Whenever a decision is enclosed in braces as the alternative of a previous decision, an unconditional transfer, within the braces, and out of the nested decision, is not treated as the end of the previous alternative.

Example 3:

```
A = B:  {C = 0: ----; ----; Q.  };
        {M = Z:  ----;  ;  };
```

Same example without the braces:

```
A = B:  C = 0:   -----;  -----;  Q.
        M = Z:   -----;  ;  ;
```

Both of the above examples generate the same code.

Up to sixteen simple comparisons may be strung together with the symbols:

$\cup$ - logical -OR-

$\cap$ - logical -AND-

In any such string only one of the logical operators may be used, i.e., no mixing of -and- and -or-.

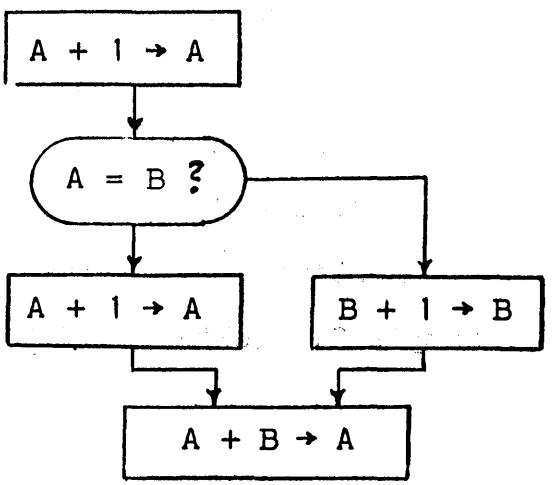
Example 4:

$A = B \cap C < D < E \cap F \neq G$:

The NELIAC equivalent of a simple flow diagram is shown in example 5.

Example 5:

```
 NELIAC
A + 1 → A,
A = B:
  B + 1 → B;
  A + 1 → A;
A + B → A,
```



C. LOOPS

Each compiler generated loop in a NELIAC program is enclosed in braces and preceded by the loop control. Loops use the register variables i through n only.

Example 1:

```
i = 10(1)20  { -------- },
```

The above example says: Set i equal to 10, execute the operations between the braces at least once, ask if i is equal to 20; if it is , clear i and ignore the brace; if not, increment i and return to the routine enclosed in the braces.

Note the difference in the basic format for an implied decrementing loop:

Example 2:

```
i = 10(1)0  { --------- },
```

This loop is executed in the same fashion as the above loop, except that i is decremented by one to zero.

Basic form of the loop control:

```
alpha = beta(gamma)delta { -------- },
```

Alpha may be any one of the register variables i through n. Beta may be an integer, a fixed point whole or half word (subscripted or not), or a register variable plus or minus an integer. Gamma must be an integer, and a minus sign must be inserted before the integer if the programmer desires a decrementing loop, unless the delta is a written zero as in Example 2. Delta is of the same form as beta.

The programmer can specify any loop increment or decrement, gamma; however, the loop will be terminated only if exact equality with delta is obtained.

D. SUBSCRIPTING

Any variable may be subscripted by an integer or a register variable or by a register variable plus or minus an integer.

Example 1:

$A[1 - 40_8] + B[2] \rightarrow C[1],$

One can also subscript without an operand.

Example 2:

$[1] + [2] \rightarrow [1 + 10],$

The above expression means the contents of the cell whose address is in 1 plus the contents of cell 2 is stored in the contents of the cell whose address is 1 + 10.

When indicating bit limits with subscripts, the subscript comes before the bit notation.

Example 3:

$A[1](11\rightarrow22) + B[j](23\rightarrow29) \rightarrow C[k](24\rightarrow29),$

Subscripted straight or return jumps may be made to jump tables in the dimensioning statement.

Example 4:

, A[j].

Example 4 executes a straight jump to the address contained in the jth element of the jump table -A-. See Section II-C.

Double subscripting in arithmetic expressions may be used when working with a two - dimensional array. Two dimensional arrays may not be used before they are defined in a dimensioning statement. See Section II-F.

In summary, subscripting may be written with register variables, integers or with register variables plus or minus integers.

Example 5:

```
A[2] →
A[1] →
A[1 - 20] →
A[k + 30]
```

The use of double subscripting on single dimension lists is meaningless and should be avoided.

The following alternate forms are legal, but should be used as sparingly as possible, since they generate less efficient code. -P- is a whole or half word.

Example 6:

```
A[P] →                    B[1, o] →
A[P + 10] →               C[k, k] →
B[P, 0] →
B[P, 1] →
B[P, P] →
```

The forms in Example 6 are not legal loop control operands.

E. SUBROUTINES AND FUNCTION DEFINITIONS

Example 1: P: { ------ },

Example 1 illustrates the definition of a subroutine P. When a call of P is made in any other part of the program, control will be shifted to the flowchart enclosed within the braces. When the logical flow comes to the right brace, control will be shifted back to the point from which P was called. A subroutine cannot be executed in any way without calling the name of the subroutine.

A function is a subroutine with associated parameters.

Example 2:

F(A, B, C,): { ------- },

In example 2, F is defined as a function with associated formal parameters A, B, and C. Those parameters are local to that function. The area between the parentheses is treated exactly as in dimensioning.

Example 3:

Function 1({A(0→5), B(10→22)}(20) = 7, 10,):

{ ------- },

Example 3 shows the use of normal dimensioning capabilities within the parentheses.

Output parameters may also be included in the function.

Example 4:

F(A, B, C; D, E): { -------- },

Example 4 illustrates 5 formal parameters: A, B, C are input

parameters; D and E are output parameters. The input parameters are separated from the output parameters by a semicolon.

Whenever the programmer writes a function with a single output parameter which he wishes to preserve in an arithmetic register, he must insure that the desired parameter is in the Q-register. (Note 2).

Example 5.

F(E): { --------, answer → answer, }

Example 5 shows a way in which the programmer can insure that the parameter will be in the Q-register by entering the whole word -answer- and immediately re-storing it. However, this is usually not necessary since most arithmetic computations leave the result in the Q-register.

It should be re-emphasized that the parameter names associated with the function definition are local to that function. These names may be used other places in the flowchart logic without danger of conflict.

All names associated with subroutines or functions are local to that subroutine or function, i.e. one cannot call on the functions formal parameter names outside the function and one cannot transfer into a subroutine or function except through the normal entrance.

Note 2: The Q-register is the auxiliary arithmetic register in the AN / USQ-20 computer.

F. FUNCTION CALLS

A function may be called by simply writing:

Example 1:

Q(F),

In example 1 the input parameter F is transmitted to the corresponding position in function Q, then the function Q is called. If there is a single output parameter, the parameter can be left in the Q-register by the function and utilized as in example 2.

Example 2:

SIN(A) → B,

NOTE: Here a function has been used as a noun or variable.

A function can be used in an expression.

Example 3:

(SIN(X) + COS(Y)) × ARCTAN(Z) → Q,

Output parameters are placed to the right of a semicolon.

Example 4:

F(A; B, C),

Example 4 says transmit parameter A to the function F, evaluate the function F; the output parameters are then transmitted to the variables B and C.

Functions may have mixed mode parameters. The programmer must insure that parameters of matching mode are set up in the correct order. If the function has been defined with more parameters than are used in the function call, the parameters will be normalized to the right (i.e. the last parameter

called will be transmitted to the right-most position in the function definition). In using both input and output parameters, all output parameters called for in the function definition must be utilized; otherwise the last input parameter called will be transmitted to an output parameter and the function call will be meaningless.

The mcde of arithmetic performed on the implicit output of a function called in an arithmetic expression is determined by the mode of the last parameter in the function call.

To reiterate, in the function call the parameter names bear no relationship to the parameter names in the function definition. The parameters used in the function call must have been defined previously in the dimensioning statement before they will compile correctly.

Example 5:

Function Definition - Absolute(A; Abs A):

Function Call - Absolute(Value[i]; Abs Value[i]),

In Example 5, the parameter -Value[i]- would be transmitted to the function Absolute and evaluated as -A-. After execution of the function, Abs A would be stored in -Abs Value[i]-.

G. MACHINE LANGUAGE

Machine language may be used at any point in the flowchart logic. Expressions of this form are completely unnecessary and anachronistic in view of the compilers ability to -declare- machine dependent functions. This notation is common to less developed NELIAC compilers and is included to aid the programmer in reading obsolescent NELIAC programs.

Example 1:

```
10 000s 0,          (clear Q-register)
26 030s a,          (add the whole word -a-)
14 030s b[j-1],     (store in the whole word b[j-1])
```

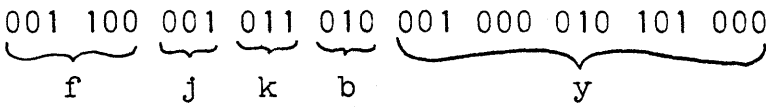
Each machine command begins with the five octal digits corresponding to the -f-, -j-, -k-, and -b- designators followed by an octal sign. At least one digit of -operand- must follow the octal sign, which is understood to be decimal unless modified with another octal sign. Named variables, with or without subscripts, are permissible as operands. Each machine command is terminated with a comma. If both subscripting and a non-zero -b- designator are written, the subscripting takes precedence.

Example 2 shows the makeup of a typical AN/USQ-20 machine instruction with its associated meaning. Also shown is the binary representation of the same instruction in core.

Example 2:

INSTRUCTION	MEANING
14 1 3 2 10250	Store the contents of the
f j k b y	Q-register in cell (10250 +
	the contents of B2) and
	skip the next instruction.

BINARY REPRESENTATION IN CORE



H. SAMPLE FLOWCHART LAYOUT

5

DIMENSIONING STATEMENT (may be omitted)

;

NAME OF ROUTINE: (may be omitted)

FLOWCHART LOGIC (may be omitted)

..stop code

IV. DECLARATIONS

A. MACHINE DEPENDENT OPERATIONS

Declarative statements are a means of putting -machine dependent- operations into NELIAC language without -machine coding-in the flowchart logic.

Input-output functions particularly need this kind of implementation. I/O functions are defined in the declaration before they are called upon in the regular NELIAC flowchart logic. This means that the declarative statement must always be read first in the line-up of flowcharts; preceding dimensions, subroutines, executive routines, etc.

There is a system declaration containing general utility routines provided with NELIAC. See Appendix C. The programmer may also provide one users declaration statement which must be in the flowchart format and kept separate from the other flowcharts. The compiler will store only one users declarative statement which must be re-declared before each compiling run.

B. ESTABLISHING LOCATIONS

Declarative operations do not allocate memory locations or produce machine language. However, the programmer can use the declarative ability of the compiler to establish the location,

for example, of the real time clock, interrupt entrances, or any machine code program which does not otherwise fit into the framework of the Neliac system.

Examples:

NAME		K Designator	ADDRESS	
CLOCK	:	3	36_8	,
SIN	:	0	140_8	,
COS	:	0	147_8	,
RTPO	:	0	32000	,
QPZ	:	2	30000	,

Each name defined in this manner is followed by a colon; the first octal digit after the colon is interpreted as a k-designator, and the rest of the number is read as an octal or decimal absolute machine location.

C. INPUT-OUTPUT SYSTEMS

The declarative statement merely describes the input or output functions. Whether the function is input or output is determined by the sense of the active statement in the flow-chart logic. The declarative I/O statements are implemented for the AN/USQ-20 computer.

The following eleven English phrases describe the input-output operations. These phrases are referred to as declarators.

1) External Function
2) Release Interrupt Lockout
3) Jump Active
4) Terminate Buffer
5) Buffer
6) Monitor Buffer
7) Generate Buffer Control Word
8) Delay
9) Machine
10) Set Int Interrupt Entrance
11) Set Ext Interrupt Entrance

Each name -declared- may refer to a specific communication channel in parantheses.

Example 1:

```
START PUNCH = (4), ----------
START FLEX  = (8), ----------
START READER = (4), ---------
```

Following the channel number, a series of mixed individual operations may be described in the following four categories.

Category 1: In order to control the operand in the flow-chart logic, Category 1 should be used in the declaration. These declarators require an operand from the flowchart to generate the appropriate function codes determined by the sense of the active input-output statement.

The declarators applicable to Category 1 are: <external function>, <release interrupt lockout>, <jump active>, <buffer>, <monitor buffer>, <generate buffer control word>, <delay>, <set int interrupt entrance>, <set ext interrupt entrance>.

Example 2:

DECLARATION

START EQUIPMENT = (4) <external function>,

FLOWCHART LOGIC

[Start Equipment <20_8>,]

Category 2: These operations indicate that no operand is taken from the flowchart logic active statement. Each of these operations is purely parenthetical, no operand is required from the active statement. The current location is used as the operand, if appropriate. The applicable declarators are:

(external function(20_8)), (release interrupt lockout), (jump active), (terminate buffer), (delay(10)), (machine code(17030_80)).

Example 3:

DECLARATION

START PUNCH = (4) (external function(20_8)),

FLOWCHART LOGIC

[Start Punch <,],

At least one quotation mark, -<-, must appear in the callout.

Category 3: These operations are used when the programmer defines one operand in the declaration and another in the flowchart logic and calls for a summation of the two operands. All operands declared must be legal fixed point numbers. The declarators applicable are:

<external function(0200000000_8)>, <machine (11030_80)>,

example 4:

DECLARATION

REWIND = <External Function(02000 00000_8)>,

FLOWCHART LOGIC

[Rewind <unit number>,],

Category 4: A declarative statement may include previously declared names, and indicates the order in which they are called.

Example 5:

RUN AMOCK = Dump, Set, E, Tape, (all previously declared)

Previously declared names may be used to set up a hierarchy of declarations. Such hierarchies are identical in principle to those declarations which consist entirely of the three basic declaration types.

The implications of the declarators which describe the input-output operations are:

1) External Function:

The external function declarator is legal in all three categories, and is used to control external equipment. Basic commands to external equipment and other computers are given with this declarator.

2) Release Interrupt Lockout:

This declarator is legal as a Category 1 or 2 function. In Category 1, a simple release interrupt jump is generated for transfer to the required operand location. In Category 2 a release interrupt instruction is generated.

3) Jump Active:

This declarator is legal in Categories 1 or 2 . When used in Category 1, the last pertinent sense, i.e. input or output, is used to generate an input or output jump active. When used in Category 2, a jump to current location is generated as in Category 1.

4) Terminate Buffer:

This declarator is legal only in Category 2 and takes the -sense- of the active statement to generate appropriate instructions to terminate the buffer.

5) Buffer and 6) Monitor Buffer:

These declarators are legal only in Category 1. They require special operands, called running subscripts, to describe what is to be buffered. The -sense- of the active statement is used to generate input or output, monitors or ordinary buffers, as appropriate.

7) Generate Buffer Control Word:

This declarator is legal only in Category 1. The operands required are identical to the Buffer or Monitor Buffer operands. The buffer control word is simply transmitted to the Q-register independent of the -sense- of the active statement.

8) Delay:

The delay generated is equal to the number of machine executions of an index jump instruction as specified by the operand + 1.

9) Machine:

This declarator is legal in Categories 2 and 3, and is described below.

10) Set Internal Interrupt Entrance:

This declarator is legal only in Category 1. It causes the compiler to generate code in the object program which transmits to the appropriate input or output internal interrupt entrance a return jump instruction to an interrupt subroutine for the particular channel defined in the declaration.

11) Set External Interrupt Entrance:

This declarator operates exactly the same as Number 10, except that the appropriate external interrupt entrance is set.

D. MACHINE CODE

A letter -L- following the numerical expressions in the lower half of a Category 2 operation indicates modification relative to the present location by the amount of that numerical expression.

A letter -k- following a machine command in a Category 3 operation indicates that the -k- designator of the operand in the active statement will suppress the -k- designator in the declaration. See Example 2.

The declarative statement is also used when it is desirable to use machine language instructions for minimizing

execution time. For example, the repeat instruction is faster than a loop for search operations and can be called up in a NELIAC flowchart by defining the operation in the declarative statement.

Example 1:

```
5(COMMENT:  FLOWCHART)
A(100₈), B,;
[SEARCH ZERO <100₈>, <A>, <NOT FOUND>, <B>, ],
```

This expression calls for a search for zero in list A, which contains 100 items. The value of the index when the zero is first located is put in cell B, and the location at which the program is to be continued if the search is unsuccessful is the verb -not found-.

The compiler implements the call-out in Example 1 by inserting the series of machine instructions in the object program which have been defined by a system declarative statement as follows:

Example 2:

```
SEARCH ZERO = <Machine Code(70230₈ 0k)>,
              <Machine Code(11437₈ 77776₈ k)>,
              <Machine Code(61000₈ 0k)>,
              <Machine Code(16730₈ 0k)>,
```

The -k- in the -y- part of the machine instruction refers to the -k- designator of the operand in the flowchart call-out. See Appendix C for a listing of the implemented system declarations.

E. ACTIVE INPUT - OUTPUT STATEMENTS

Each active input-output statement will generate a variable amount of code compiled as an open subroutine, that is, the code will be inserted each time the statement is written. If the programmer wishes to obtain the code only once, he should enclose the active statement in braces to make it into a closed subroutine. The closed subroutine may have the same name as the I/O statement, and is called as an ordinary subroutine.

Input-output statements are similar to function calls in that the programmer must make his operands line up with the I/O declaration.

The -sense- of the statement, that is, input or output, is determined by the -quotation- operators.

Example 1:

>input< <output>

Each statement begins with unique current operator-next operator pairs of [A< or [A>. The name A must be defined as an input-output function name. Input operands may be mixed with output operands.

Each operand must be enclosed in a set of -quotation- operators. Commas are used to separate the operands.

Part of a list may be used as the operand of an active statement.

Example 2:

```
DECLARATION
        A = (4) <buffer>,
FLOWCHART
        [ A <B[C→D]>, ],
```

The active statement says, with reference to the declaration: Initiate an output buffer on channel 4 of the area starting with the element B[C] through the element B[D]. The running subscripts may be integers, register variables, or fixed whole or half words.

Example 3:

```
[ A <B>, ],
```

Example 3 says: Initiate an output buffer on Channel 4 to output the whole of list B.

Example 4:

```
[ A <[B]>, ],
```

Example 4 says: Initiate a buffer using B as the buffer control word.

Example 5:

```
5(COMMENT:  DECLARATION)
PRINT = (4) <external function>
<external function(200000000s)>,
<buffer> (jump active)..
```

```
5(COMMENT:  FLOWCHART)

A(10), status;
[print> status <, <i>, <A>,  ]..
```

If the flowchart in this example were compiled at cell 10100, the machine code generated would be:

```
10000 61000 10113
10100 00000 00000      (a)
10112 00000 00000      (status)
                       external function
10113 17230 10112
10114 10101 00000
10115 02000 00000
10116 26030 10115
                       external function (200000000)
10117 14130 10120
10120 00000 00000
10121 13230 10120
10122 20100 00000
10123 10111 10100      buffer
10124 74230 10123
10125 63200 10125      jump active
10126 61400 10000
```

Example 6:

Hypothetical input problem

There are three logical records on each block of tape. Each record is 100 locations long. Search word for the block is -3 MAY- in compiler code, left justified. Read the block, inserting the logical records into three discontinuous areas using the logical tape unit i.

DECLARATION

```
5
READ 3 RECORDS = (10)
<external function(4600000000₈)>,
<external function>,
<buffer> (jump active),
<buffer> (jump active),
<buffer> (jump active)..
```

(FLOWCHART)

```
5
R(100), Q(100), P(100),  [search word: 3 may];
NAME:
[read 3 records <i>, <search word[0]>,
>P<, >Q<, >R<, ]..
```

V. OPERATORS GUIDE

A. CARD INPUT

NELIAC uses only the first 72 columns of each card. The programmer may insert card numbers or any other information he desires in columns 73 through 80. When the compiler edits the flowcharts for output, new card numbers are assigned. Columns 73 through 77 of the output cards are punched with the flowcharts sequence number, columns 78 through 80 are used for flowchart line numbers (line numbers increment by three, leaving two unused line numbers for every card). The use of the editing routine of NELIAC is highly recommended, since many logical errors can be discovered by examining the spacing and indentation of the output flowcharts.

B. LOAD NUMBERS

Each flowchart begins with one of the ten load numbers. The function of the flowchart in the system is uniquely described by that load number.

0 - Flowchart plus the edited output of that flowchart.

1 - Declaration plus the edited output of that flowchart.

2 - One line correction plus the edited output of that flowchart.

3 - Flowchart correction plus the edited output of that flowchart.

4 - Executive flowchart plus the edited output of that flowchart.

5 - Flowchart

6 - Declaration

7 - One line correction

8 - Flowchart correction

9 - Executive flowchart

C. CORRECTION LOADS

Each correction flowchart must have a sequence number associated with it. The format for a correction is:

Load Number

Sequence Number

Correction Load

When a single line correction is made, the following format must be strictly observed:

Example 1:

7 or 2

Sequence number

First comparison of at least 10 characters

Correction line

Second comparison line of at least 10 characters.

When using paper tape input, each line is followed by a carriage return with a stop code at the end of the last line. When using card input each line must be on a separate card. The comparison lines are composed of a string of characters which are independent of the original spacing and indentation of the flowchart. Spaces in alpha-numerics are significant, however, and must be duplicated. The correction line may be blank, but must always exist.

Flowchart corrections are made to replace an entire flowchart on the input flowchart tape.

Example 2:

8 or 3
Sequence number
5 or 6
(FLOWCHART)

When using paper tape input, a stop code follows the the sequence number on the leader of a 5 or 6 load flowchart. When using card input, the load number and sequence number must be on separate cards preceding the 5 or 6 load flowchart.

D. BATCHING CORRECTIONS

All corrections must be -batched-, i.e. all sequence numbers of correction loads must be greater than or equal to the sequence numbers of previously loaded corrections. If this rule is not observed, additional passes must be made to update the input master flowchart tape.

E. FUNCTIONS OF THE 9 LOAD

The 9 load, or executive flowchart, always indicates the end of the loading and correcting phase of the compiling process. At present, the information in this flowchart should be the programmers name and the date.

F. NELOS

NELOS (Navy Electronics Laboratory Operating System) reflects an operating philosophy necessary for the generation and checkout of large-scale programs whose characteristics make them difficult to handle with less powerful tools. The NELOS philosophy is simple: NELOS provides automated supervisory control over the NELIAC compiler, over program execution, and over the utility programs in the system. Supervisory control must necessarily be informed control; to this end, sections describing the operation of the NELIAC compiler and of the monitor are included below.

NELOS has three basic parts: (1) The executive program; (2) The monitor; and (3) The utility system. The executive program controls the execution of all programs in NELOS. The monitor program is a debugging aid which can interpretatively execute programs, insert and delete dynamic core dumps, and provide for dynamic source language data introduction. The utility system is a library of often used programs such as core dumps, tape dumps, tape copy programs, etc., which can be called in and executed under NELOS control.

The NELOS executive program does a limited amount of automatic sequencing during the compiling process. On the whole, NELOS is controlled by the set of fourteen NELOS operators. Any collection of these is called a NELOS control statement. It is a mistake to assume that there are only fourteen basic functions in NELOS, however, since the operators can be used in any combination to perform uniquely different tasks.

The "define segment" operator gives NELOS the ability to generate programs which are too large to fit into memory at once and must be split into pieces or "segments" which are called in from auxiliary storage for execution. NELOS is specifically designed to aid in the production of these programs from the NELIAC language. Because of the close interrelationships of NELOS and the NELIAC compiler, a thorough understanding of the operating principles of NELIAC is a prerequisite for the successful use of NELOS.

G. NELIAC OPERATING CHARACTERISTICS

The basic input element to NELIAC is called the "flowchart." Any collection of NELIAC or NELOS statements grouped together is always called a "flowchart."

Identification of input types is effected by load numbers (i.e., the first character in a flowchart is taken to be the load number. If the first character in the flowchart is not a number, that flowchart is identified as a NELOS control statement.).

The flowcharts are stacked for input to NELOS in the following order:

Initial control statement

Declaration

NELIAC flowcharts

Terminal control statement

The first flowchart must necessarily be a NELOS control statement and is called the "initial" control statement. This control statement is

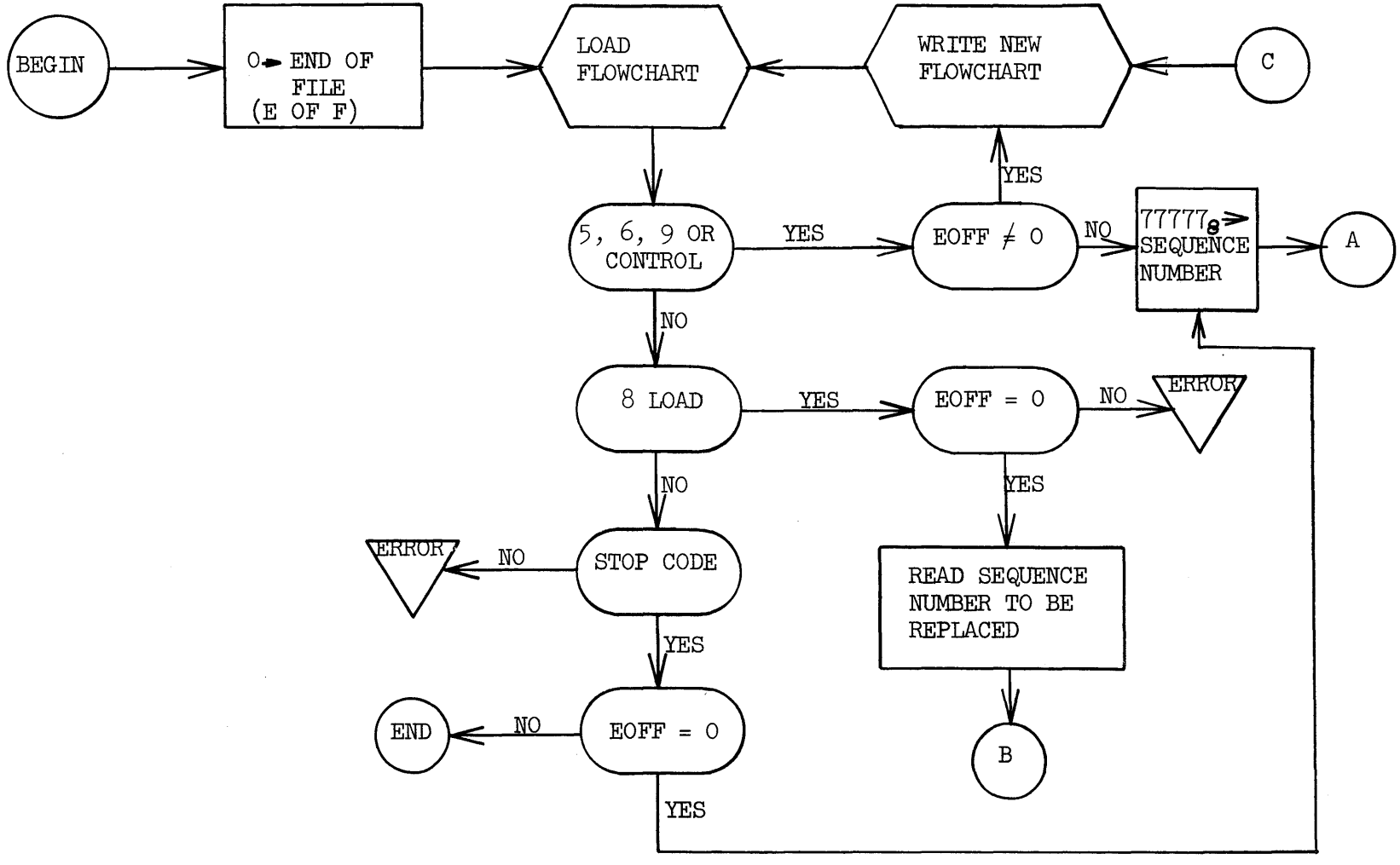
different from other control statements in that it must be submitted for every NELOS run, unlike other control statements which are preserved on the master flowchart tape, and need not be resubmitted.

The NELIAC compiler has three separate operating phases:

1. Update and code conversion
2. Compiling
3. Output

1. The update and code conversion phase does a very simple job; it merely takes the source language input element (the flowchart) and converts the card codes to a set of NELIAC compiler codes. The code conversion program (load flowcharts) converts the hardware codes for "Begin" to "{" and "End" to "}", etc. The update phase places the binary core image of the NELIAC compiler codes onto the output flowchart tape for subsequent use by the compile phase. By the use of correction load numbers, a flowchart image may be replaced on the output flowchart tape.

Figure 1 is a detailed flow diagram of the operation of the update phase. "EOFF" is "end of file flag on old flowchart tape." "STOP CODE" is the hardware stop code recorded on the card image tape. When an error is detected, control shifts back to NELOS with an appropriate error message. Note, corrections must be batched, i.e., correction numbers must be in an ascending sequence. Every flowchart is counted in the correction process, the sequence numbers are implicit, i.e., the first flowchart is number 1, the second is number 2, etc. New flowcharts to be added to a program are always recorded after all the old flowcharts.



UPDATE CYCLE

FIGURE 1A

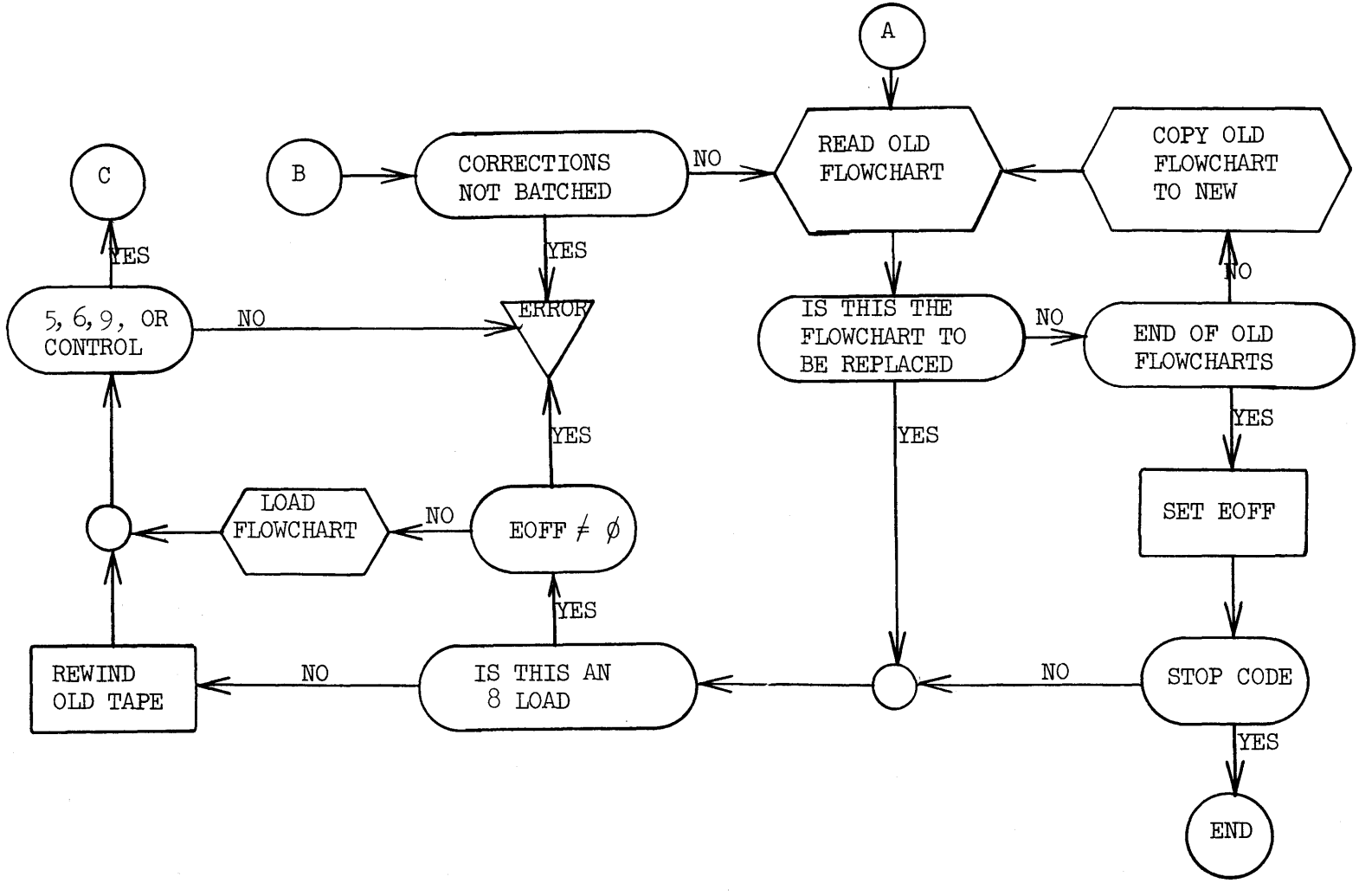

A
READ OLD FLOWCHART
COPY OLD FLOWCHART TO NEW
B
CORRECTIONS NOT BATCHED
NO
YES
C
YES
5, 6, 9, OR CONTROL
NO
ERROR
IS THIS THE FLOWCHART TO BE REPLACED
NO
END OF OLD FLOWCHARTS
NO
YES
YES
YES
LOAD FLOWCHART
NO
EOFF ≠ ∅
SET EOFF
YES
REWIND OLD TAPE
NO
IS THIS AN 8 LOAD
NO
STOP CODE
YES
END

FIGURE 1B

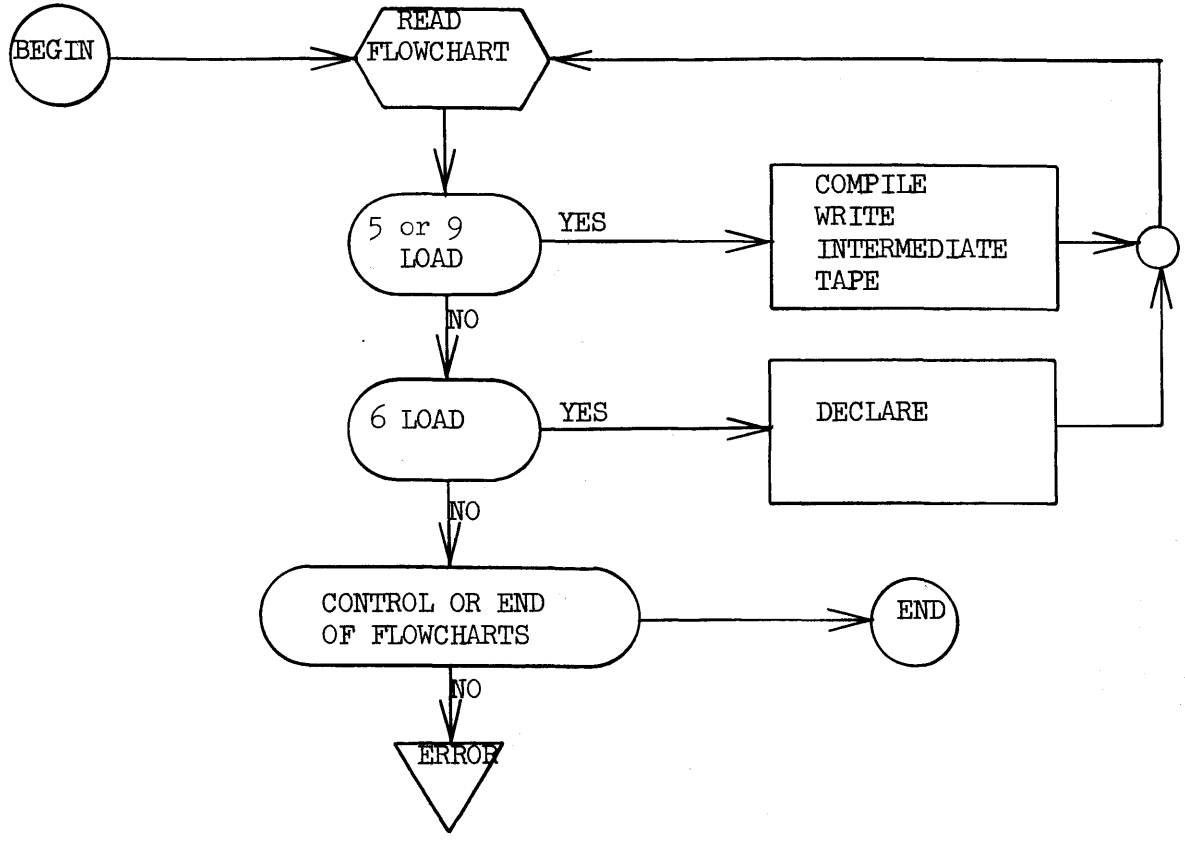
2. The compiling process is a one-pass operation; the source language statements are read, and machine language produced without any intervening intermediate language or assembly phases. When the process is complete for one flowchart element, all undefined or "future" references are written out on a scratch tape with the machine language produced from that flowchart. See Figure 2.

3. The output phase of NELIAC uses the intermediate output tape of the compile phase to generate a binary object program tape which contains the machine language in NELIAC format.

In addition, the output phase contains the formatting programs which generate the name list dump, the object program dump, and others. The output listing tape contains a list of all syntactical errors detected during the compile phase and a list of all undefined names detected during the output phase. A listing of the names of the flowcharts compiled in sequence, along with their running locations and entrances, if any, is also produced.

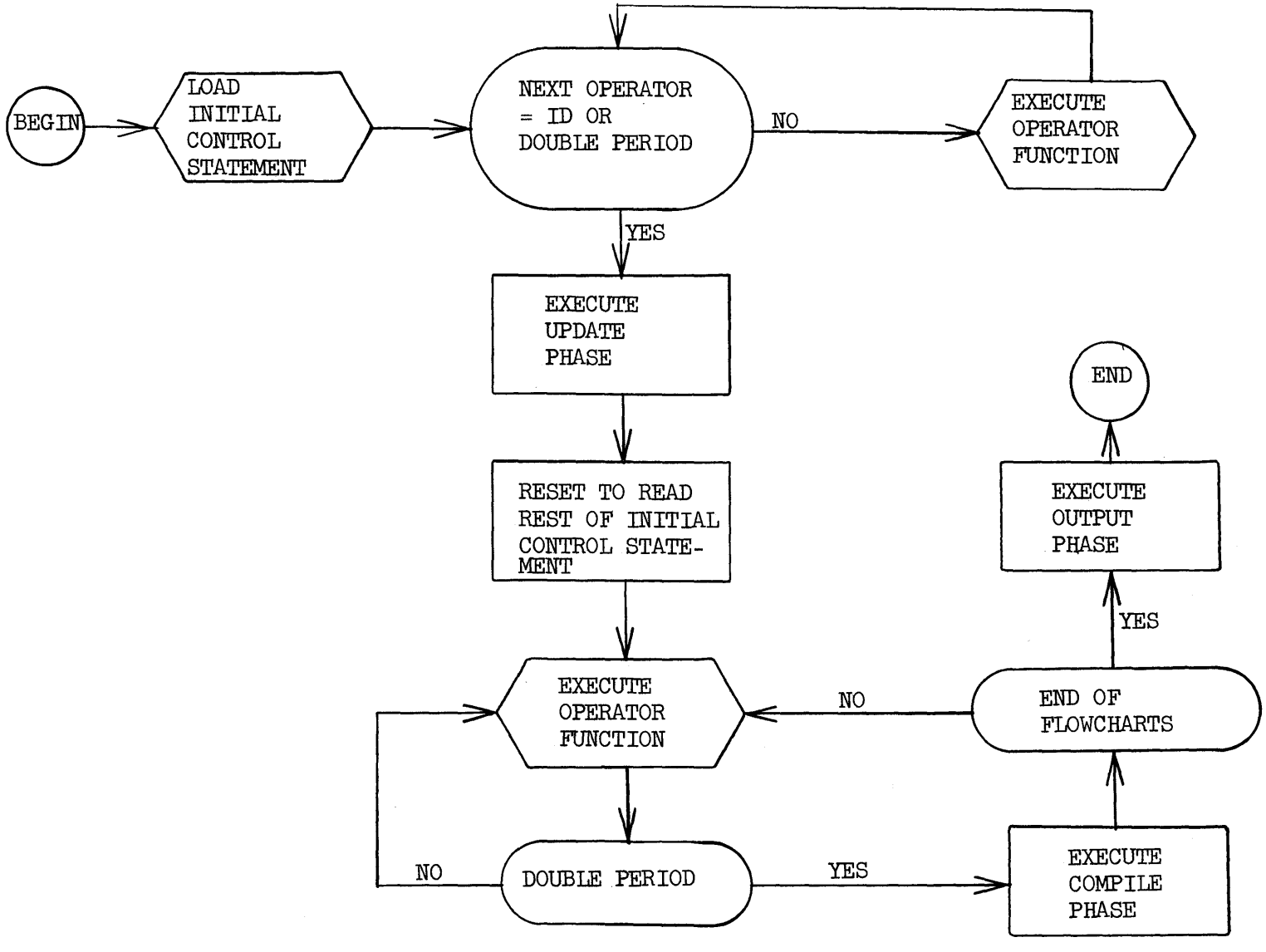
H. NELOS EXECUTIVE PROGRAM

The executive program reads the NELOS operator and executes the functions necessary to accomplish the required task. The first or "initial" control statement has a unique function since it is not placed on the flowchart tape with the other flowcharts introduced into the system. The reader should examine Figure 3 and Figure 4 for an explanation of the automatic sequencing of NELOS. Subsequent control statements are written on the flowchart tape and are handled exactly like any other flowchart. Note Figure 2.



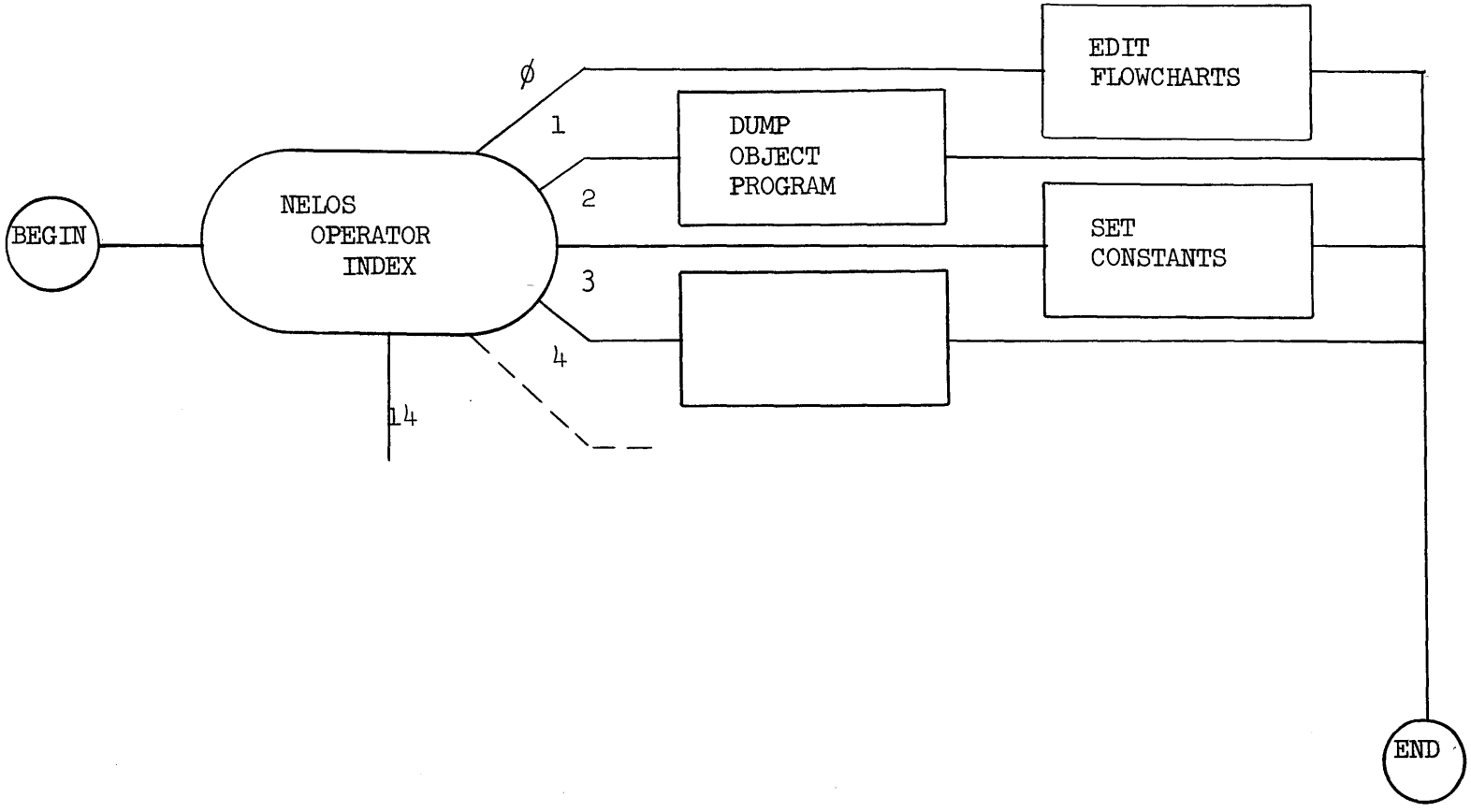
COMPILE PHASE

FIGURE 2



NELOS AUTOMATIC SEQUENCING OF THREE NELIAC PHASES

FIGURE 3



EXECUTE OPERATOR FUNCTION

FIGURE 4

The reader should observe that NELOS control statements can be executed while compiling, and that control can be exercised between each NELIAC flowchart.

A very important part of NELOS is the "ENVIRONMENT GENERATION" program. This program produces a tape which contains NELIAC's "NAME LIST" or "ENVIRONMENT DESCRIPTION TABLE" and the machine language of the programs whose names make up the name list. Every name defined at the time that the environment tape is generated is included; all programs and allocated tables are also included. Note, however, that nonallocated tables do not exist as zeros on the tape, but only the pertinent table and item names are preserved. Any NELIAC program may become part of the "ENVIRONMENT" stored on tape. This tape will later become an input tape to NELOS; whenever it is called, NELIAC will recall all those previously defined names and programs. If a new environment tape is then generated, an augmented, enhanced environment will be the result. The new environment is indistinguishable from an environment generated with all the flowcharts compiled at one time. Since it is not possible to change core allocation addresses on the environment tape, it is recommended that a "Master Flowchart" tape be generated and continually updated for this specific purpose by the system user. If such a tape is generated, then the flexibility of NELIAC correction loads is extended to the new environment tape.

The tables and programs defined on the environment tape are typically those designed to work with programs which are not necessarily in memory at the same instant. Commonly used subroutines or utility

and input/output packages would also be likely residents of the environment tape. The value of the "environment tape" lies not so much with the saving of compile time for often-used programs, but in the clarification of the status of such programs and tables for each individual contributor to a system program. Naturally, the burden of currency is placed on the individuals responsible for the maintenance of the environment tape.

I. THE NELOS OPERATORS

The NELOS operators discussed in this section are written in control statements just as they appear as the headings of the explanatory paragraphs. If a nonexistent operator is written, or if the form is not strictly observed, a NELOS error message will appear and the run will be abandoned.

There are three categories of NELOS operators:

(1) PASSIVE

ID: PRINT: STOP, SET CONSTANTS (BEGINNING ADDRESS, ENDING ADDRESS),

(2) OUTPUT

EDIT FLOWCHARTS (FIRST FLOWCHART, LAST FLOWCHART), DUMP OBJECT PROGRAM (FIRST FLOWCHART, LAST FLOWCHART), DUMP NAME LIST,

(3) ACTIVE

DEFINE SEGMENT, TERMINATE RUN, CALL ENVIRONMENT, INCLUDE MONITOR, EXECUTE, GENERATE ENVIRONMENT, CALL UTILITY (N,,),

The "passive" operators can be written in any control statement. Their use does not affect the operating characteristics of NELOS in any way, i.e., the normal sequence of program generation will be followed as illustrated in Figure 5.

The "output" operators can be used only at the termination of one of the major compiling phases. NELOS must reference some of the working tapes in the system to accomplish the tasks related to output operators, so their use is restricted to times when the tapes are being passed from one phase of NELOS to the next (such as the end of the update cycle or when a new environment tape is generated).

The "active" statements define the operating characteristics of each NELOS run. The active statements may be written in any control statement, but some understanding of the NELIAC compiling procedures is essential for meaningful operation.

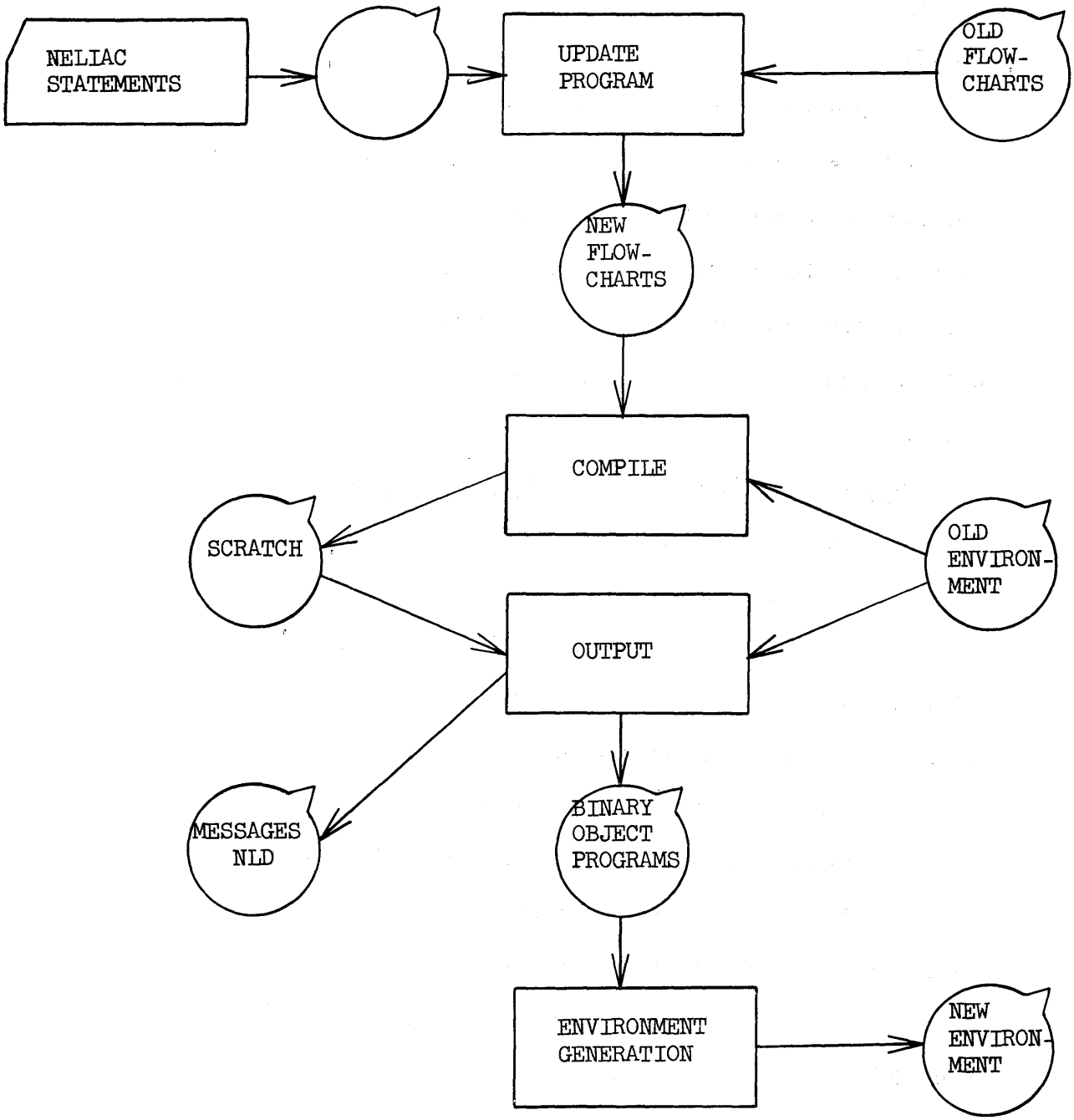
The implications of the NELOS operators are described in the following paragraphs:

ID:

The ID, or identification operator may be written in any control statement. The "ID" may be any alphanumeric string not exceeding 50 characters and is terminated by a period. The alphanumeric information is preserved in NELOS and is used on all the outputs generated until a new ID is given. This operator also signals the beginning of the update cycle (see Figures 1A and 1B).

PRINT:

The print operator causes the alphanumeric string following the colon to be displayed on the supervisory printer. The string is



PROGRAM GENERATION CYCLE

FIGURE 5

restricted to a maximum of 50 characters. A period terminates the string.

STOP,

The stop operator halts the computer until the high speed switch is depressed. The stop statement is used typically to allow the computer operator to mount or dismount tapes following a request to do so as the result of a previous "PRINT" statement. The computer operator may reassign NELOS' logical tape drives at this time to utilize a newly generated environment or to swap malfunctioning tape units.

SET CONSTANTS (BEGINNING ADDRESS, ENDING ADDRESS),

The set constants operator allows the programmer to allocate his program to specific core locations starting with the octal or decimal locations written as "BEGINNING ADDRESS." The "ENDING ADDRESS" index is a threshold indicator to the compiler which causes a message to be placed on the error tape when the threshold is passed. See Sample Control Statements, page V-21.

EDIT FLOWCHARTS (FIRST FLOWCHART, LAST FLOWCHART),

This operator can only be used in the initial control statement, and is always executed at the end of the update cycle. The inclusive flowcharts specified by "BEGINNING ADDRESS" and "ENDING ADDRESS" are dumped on tape for subsequent listing and card punching.

DUMP OBJECT PROGRAM (FIRST FLOWCHART, LAST FLOWCHART),

This operator is legal only in the last or "terminal" control statement, since NELIAC must perform the output phase to execute this operator, and therefore terminates the compile phase.

DUMP NAME LIST,

This operator produces a cross-referenced output of the tags, labels, verbs, and nouns used previously in the compiled program, whether included with the environment or generated by the object program.

DEFINE SEGMENT,

This operator is legal throughout the compiling phase. "DEFINE SEGMENT" will cause the program segment delimited by "DEFINE SEGMENT" operators to be stacked on the object program tape. This operator is typically followed by a "SET CONSTANTS" operator, although it need not be so followed.

TERMINATE RUN,

This operator is legal in any control statement, and simply calls the next job. The use of this operator is mandatory when the job to be done does not involve compiling. See Figure 3.

CALL ENVIRONMENT,

This operator should be used only in compiling control statements, and causes the load of the environment description table (name list) into NELIAC's working storage. The reader should observe that "CALL ENVIRONMENT" will cause the compiler to forget all names not in the environment description table.

INCLUDE MONITOR,

This statement must be written if any calls to "ENTER MONITOR MODE" are used in the 9-load flowchart (see Section V-K). The monitor program is recorded as the first program on the object program tape and appears only once in the core program produced.

EXECUTE (N),

The "EXECUTE" operator is used to run a program which is on the object program tape. "N" is an integer which specifies the number of program segments to be loaded from the tape. The last program segment loaded will be executed first. "N" can also be computed as the number of times "DEFINE SEGMENT" is written + 1.

GENERATE ENVIRONMENT,

This operator causes NELOS to generate an environment tape on the tape unit allocated for that purpose. Generation of the environment tape should be done only in the last or "terminal" control statement, since NELIAC must perform the output phase to execute this operator.

CALL UTILITY (N, A, B, C, . . . Z)

The call utility operator is used to set up a parameter string for one of the NELOS utility programs. "N" is an integer specifying the file number of the utility program desired. "A" through "Z" are integers which are input parameters to the individual utility program.

J. NELOS CONTROL STATEMENTS

Each control statement must have at least one NELOS operator and must end with a double period.

The proper use of NELOS is dependent on an understanding of NELIAC compiling techniques. It is obviously inadvisable to attempt such things as an object program dump or environment generation before

compiling.

Given below are some sample control statements. These are given to acquaint the student with the appropriateness of most combinations.

SAMPLE CONTROL STATEMENTS

```
ID: P, SMITH 7 DEC 62.
EDIT FLOWCHARTS (3, 4), CALL
    UTILITY (3, 7),                    Appropriate in initial
PRINT: BEGIN COMPILING, REMOVE         control statement to
    OLD FLOWCHART TAPE.                control update phase
STOP, TERMINATE RUN,
SET CONSTANTS (10000_8, 73261_8),

ID: A,GZZORK.
SET CONSTANTS (73262_8, 77777_8),
CALL ENVIRONMENT,
DEFINE SEGMENT,                        Appropriate in compiling
TERMINATE RUN,                         control statements
PRINT: TERMINATE RUN IF ANY
    ERRORS THIS FAR.
STOP,
INCLUDE MONITOR,
```

DUMP NAME LIST,	
DUMP OBJECT PROGRAM (10, 11),	Appropriate in last state-
GENERATE ENVIRONMENT TAPE,	ment. Associated with
PRINT: LABEL NEW ENVIRONMENT	compiling or output phase
TAPE: QP2 7 DEC 62.	
CALL UTILITY (2, 38, 5, 3),	
CALL UTILITY (3, 14),	
CALL ENVIRONMENT,	Appropriate as individual
DUMP NAME LIST,	control statement which
TERMINATE RUN,	does not use NELIAC
EXECUTE (3),	Appropriate as control statement which simply runs a pre-compiled program typically under NELOS monitor control

K. THE NELOS MONITOR

The NELOS Monitor Mode of operation, a debugging aid, allows the programmer to monitor, or "watch," the operation of specified areas of his programs, through the use of his own "Monitoring Subroutines." The monitor will cause entry to the programmer's Monitoring Subroutine after the execution of each individual instruction within the program area being monitored. The monitor will continue full

monitoring of jumps (and return jumps) to instructions outside of the monitored area, unless the programmer chooses not to do so.

Upon completion of a monitored area, the monitor mode will cause entry, if the programmer elects, to his own "End-Monitor" subroutine, before continuing along with the main program.

The programmer may choose beforehand to have a monitor canceled at any time during the run. If this is to be done during the execution of the monitor to be canceled, the associated End-Monitor subroutine, if any, will be executed as a part of the canceling sequence.

Basically, then, the Monitor Mode is a vehicle for the programmer's convenience, one that leaves him wide latitude in which to program his own monitoring subroutines.

While the monitoring of an area is in progress, the address of the individual instruction being monitored is available to the programmer as the noun "P," so that the programmer may use his monitoring subroutine to save "P" and other data. When control is transferred to the programmer's monitoring subroutine, all registers will have been restored to the values dictated by the program being monitored.

ESTABLISHING A MONITOR

Monitors are established by use of the function call

ENTER MONITOR MODE (A, B, C, D, E), where:

A = The address (label) of the first instruction of the area to be monitored, (may <u>not</u> be the entry cell of a function or subroutine).

B = The address (label of the last instruction of the area to be monitored;

C = The address (label) of the programmer's "Monitoring Subroutine" * itself;

D = The address (label) of the programmer's "End-Monitor Subroutine," * if desired, or zero, if no ending routine is wanted with this monitor; and,

E = Zero, indicating the programmer wishes to monitor all instructions encountered while in the monitor mode; or, E equal nonzero, indicating the programmer has elected the option to abort full monitoring of jumps (and return jumps) to instructions outside the area to be monitored.

If parameter E is nonzero, the monitor mode retains control in the event that a jump or return jump outside the area is encountered, but does not cause entry to the programmer's subroutine until and unless the running program returns to the monitored area. (This condition is termed a "psuedo-monitor.")

ESTABLISHING AN END MONITOR

See parameter "D" under "Establishing a Monitor."

CANCELING A MONITOR

Monitors are canceled by using the function call

* Must be written as a subroutine.

CANCEL MONITOR MODE (M_1, M_2, - - - M_8).

where:

M_i = parameter "A" of an ENTER MONITOR MODE previously established. From one to eight monitors may be canceled at one time. If the Monitor Mode is unable to find a parameter "A" corresponding to M_i, the operation is ignored for that M_i.

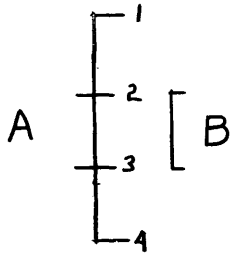
If no parameters (M's) are given in the function call, the Monitor Mode will assume that the monitor currently being executed is to be canceled immediately, and will proceed directly to the related End-Monitor Subroutine, if any. If no parameters are specified, and a Monitor Mode is not currently being executed, the operation will be ignored and the run will continue.

MULTIPLE MONITORS

Up to eight monitors may be established before any are canceled, but to establish subsequent additional monitors, one existing monitor must be canceled for each new one entered. Attempts to exceed the limit will be ignored.

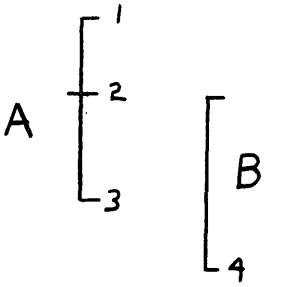
No matter how many monitors are established, only one may be in effect at any one time. With care, monitored areas may be nested, or overlapped. If, while in monitor mode A, an entry to monitor mode B is encountered, monitor mode A will only temporarily relinquish control to monitor mode B, and monitor mode B will, upon completion, automatically return control to monitor mode A. This is ideal for nesting, but requires care in overlapping.

Nested Monitor Modes



Monitoring will be done in Mode A from points 1 through 2, in Mode B from points 2 through 3, in Mode A from points 3 through 4. After point 4 the program will resume operation in high-speed mode.

Overlapping Monitor Modes



Monitoring will be done in Mode A from point 1 to point 2, in Mode B from points 2 through 4, and in Mode A again from point 4 until and unless the program happens to return to a point between points 1 and 3. Mode A cannot be terminated (unless canceled) until the instruction at point 3 is actually executed.

Since a monitoring mode must naturally cause some expansion of program running time (see "TIMING") it is important that the programmer carefully defines his limits, and builds efficient monitoring subroutines. By keeping the monitoring subroutines small and fast, and using end-monitor subroutines for time-consuming operations, such as output operations, the programmer can keep running time at a minimum. For example, it may be possible to use the Monitoring Subroutine to make a few quick entries into a buffer, and then output only once the monitored results for the entire pass through the monitored area during the End-Monitor routine.

It should be noted that the program being monitored will run at normal computer speed except while operating within a monitored area. If the abort jump option is not taken, and jumps and subroutines are subsequently monitored, more time yet will be required. (This may be desired, of course.) If the abort jump option is taken, the monitor mode continues in a "psuedo-monitor" condition, wherein the programmer's monitoring subroutine is not entered after each individual instruction is executed.

Additionally, care must be taken that the first instruction in an area will be executed, else the monitor may not be established. For instance, if the first instruction of the area is skipped, or if a jump is made into the middle of an area to be monitored, then no monitor control will be initiated. Likewise, if the last instruction to be monitored is skipped, the monitor will not be discontinued at the proper point. The method used in establishing the monitor (internally) demands this care. The monitoring program itself is initiated when an Enter Monitor Mode statement is encountered. Establishing a mode at this point, the monitoring routine saves the first instruction of the area to be monitored, and replaces it with an entry to the monitor program. Control is returned to the program under study until the monitor area is reached.

The use of the 9-load program is restricted to the writing of monitoring subroutines, formatted output or test data introduction. Since the 9-load flowchart is basically like any NELIAC flowchart, full cognizance of source language names is achieved. The separate

load number for this flowchart is necessary because of the usage of this program rather than any unique treatment by NELIAC. When the "execute" operator of NELOS is used, the execution of the 9-load program will always supersede the normal execution of the program. In this way, the monitor locations can be established for debugging runs, and ignored for production runs (as a manual option) without modifying the program in any way. When running a program in the monitor mode, care must be exercised in not disturbing the test environment defined in the 9-load program. Data may be introduced into tables in a very straightforward fashion by simply writing a NELIAC assignment statement.

Example:

100 → alpha, 200 → beta,

But, if one writes:

i=10(1) 100{i → table alpha [i] ,}

the 9-load program writer must insure that the value of "i" can safely be altered or that zero is desired as the correct value of "i" upon exit from the loop.

For more ambitious test data introduction the programmer may use a prepared test data tape and write a simple I/O statement into the 9-load program.

L. CAPABILITIES OF THE MONITOR PROGRAM

SAMPLE MONITOR PROGRAM

```
5                                                                    1
A(100), B, C;                                                        2
D:                                                                   3
J=3(1) 10 { 0 → A [i] } ,                                            4
E:                                                                   5
i=10(1)98 { A [i] + B + A [i + 1] → A [i] } ,                        6
F: ..                                                                7
9                                                                    8
T S, [g:88888|3|88888/];                                             9
Enter monitor mode (E, F, H, 0, 0),                                  10
0 → B, 10 → C,                                                       11
E.                                                                   12
H:                                                                   13
{i=j: write (g, P, i), [call NELOS < ,] ;;} ..                       14
```

The above trivial 9-load program shows the introduction of test data in source language (line 11). The monitoring subroutine is written to detect the specific case when i is equal to j during the execution of the program between the labels E and F. As a result of running this program we would have written on the NELOS message tape the octal address of label "F" and the value of "i" when

equality with "j" was obtained, which is zero. Why? This example points out the fact that the NELOS monitor is a source language debugging aid which has pointed out an eccentricity of the machine as reflected in NELIAC, but without going back to the machine language to discover it. The power of the NELOS monitor is in the fact that its focus can be broadened or narrowed as necessary to follow general or specific problems which are difficult to diagnose from static core dumps. The monitor mode may be used in its narrowest scope (i.e., beginning label = end label) to plant a dynamic core dump or test data introduction point without altering its running characteristics significantly.

VI. PROGRAMMING TECHNIQUES

A. SAMPLE PROGRAMS

Efficient programming in any language is dependent upon the programmers knowledge of the problem at hand and the techniques used in generating machine code from the language in which the program is written. There are both efficient and inefficient programming methods in any language, whether it be machine code or the most sophisticated higher level language. This Section will attempt to provide examples of NELIAC programming techniques which do provide efficient machine code for the AN/USQ-20 computer.

Example 1 is a complete program producing a table of values for a simple function. The production of the table is aided by use of the -write- package (See Section VI-B).

Example 1:

$$F(A,B) = \frac{a(b - 3.994)}{b + a}$$

Evaluate the above function over the range of A = 0.0 to A = 5.0 in steps of 0.2, B = 1.0 to B = 2.0 in steps of 0.2. Set up a table of the function for these particular values.

The table should have a heading to appear as in the example below. Each answer should have two digits to the left of the decimal point and three to the right.

table of function f

b

a	1.0	1.2	1.4	1.6	1.8	2.0
0.0	xx.xxx	xx.xxx	xx.xxx	xx.xxx	xx.xxx	xx.xxx

```
5 (COMMENT:  DIMENSIONING)

[heading: |25| < table of function f > /// |34| <b> //
|3| <a> |4| <1.0> |6| <1.2> |6| <1.4>  |6|  <1.6>  |6|
<1.8>  |6|  <2.0>  //],
[line: 0.0 (6: |2| 00.000) / ],
ans(6).  real i. real j. ;

PROGRAM: (COMMENT: FLOWCHART LOGIC)
enable paper tape,                                          1
write (heading),                                            2
i = 0(2)50 {i → real i / 10.0 → real i,                     3
j = 10(2)20 {j → real j / 10.0 → real j,                    4
(j - 10) / 2  → k,                                          5
f (real i, real j) → ans[k],}                               6
write(line, real i, ans[0], ans[1], ans[2],
            ans[3], ans[4], ans[5]), },                     7
disable paper tape,                                         8

F(a. b.):                                                   9
{a × (b × b - 3.994) / (b + a) → b, }..                     10
```

COMMENTS ON PROGRAM

LINE	
1	Call the enabling subroutine for -write-.
2	Call to write heading, which requires no parameters.
3,4	Loop control with necessarily fixed point loop control, real i- and -real j- contain computed floating representation of the indexing variables i, j. Note the automatic conversion to floating point mode of the fixed point variables i and j.
5	Compute subscript storage.
6	Call the function -f- with real i, real j, and store the result in -ans[k], -, then end minor j loop.
7	-Write- call with 7 parameters. End i loop.
8	Disable paper tape, disabling subroutine for -write-.
9	Definition of -f- with two floating point formal parameters, -a- and -b-.
10	Computation of -f- with answer in arithmetic register. (always true for floating point pseudo accumulators).

table of function f

a	b = 1.0	1.2	1.4	1.6	1.8	2.0
0.0	0.000	0.000	0.000	0.000	0.000	0.000
0.2	- 0.498	- 0.364	- 0.254	- 0.159	- 0.075	0.000
0.4	- 0.855	- 0.638	- 0.451	- 0.286	- 0.137	0.001
0.6	- 1.122	- 0.851	- 0.610	- 0.391	- 0.188	0.001
0.8	- 1.330	- 1.021	- 0.739	- 0.477	- 0.231	0.001
1.0	- 1.496	- 1.160	- 0.847	- 0.551	- 0.269	0.002
1.2	- 1.633	- 1.276	- 0.938	- 0.614	- 0.301	0.002
1.3	- 1.746	- 1.375	- 1.016	- 0.669	- 0.329	0.002
1.6	- 1.842	- 1.459	- 1.084	- 0.716	- 0.354	0.002
1.8	- 1.924	- 1.532	- 1.144	- 0.759	- 0.376	0.002
2.0	- 1.995	- 1.596	- 1.196	- 0.796	- 0.396	0.003
2.2	- 2.058	- 1.652	- 1.242	- 0.830	- 0.414	0.003
2.4	- 2.113	- 1.702	- 1.284	- 0.860	- 0.430	0.003
2.6	- 2.162	- 1.747	- 1.322	- 0.887	- 0.445	0.003
2.7	- 2.206	- 1.787	- 1.355	- 0.912	- 0.458	0.003
3.0	- 2.245	- 1.824	- 1.386	- 0.935	- 0.471	0.003
3.2	- 2.281	- 1.857	- 1.414	- 0.955	- 0.482	0.003
3.4	- 2.313	- 1.887	- 1.440	- 0.975	- 0.492	0.003
3.6	- 2.343	- 1.915	- 1.464	- 0.992	- 0.502	0.003
3.8	- 2.370	- 1.941	- 1.486	- 1.009	- 0.511	0.003
4.0	- 2.395	- 1.964	- 1.506	- 1.024	- 0.519	0.004
4.2	- 2.418	- 1.986	- 1.525	- 1.038	- 0.527	0.004
4.4	- 2.439	- 2.006	- 1.543	- 1.051	- 0.535	0.004
4.5	- 2.459	- 2.025	- 1.559	- 1.063	- 0.541	0.004
4.8	- 2.477	- 2.043	- 1.574	- 1.075	- 0.548	0.004
5.0	- 2.494	- 2.059	- 1.589	- 1.086	- 0.554	0.004

B. The NELIAC -WRITE- PACKAGE

DESCRIPTION:

The -write- package is a general purpose output package, written in NELIAC, which is in wide use for formatting, report writing and scientific output. -Write- uses the philosophy of an external-format-statement, written as a literal in a NELIAC dimensioning statement. Each -Write- call is written with its associated format literal name, and the necessary parameters to be justified, converted and formatted for output. The formats are written as -pictures- of the desired output; the -write- package scans the format literal for each parameter, and assembles the external equipment codes for whichever piece of equipment is -enabled- at that time.

Example 1:

```
Write(number format, number),
```

Example 1 illustrates a simple -write- call with the format plus one parameter. All calls on -write- are made in a similar fashion. Up to thirty output parameters may be handled with a single call.

PICTURES:

Example 2:

```
8888
```

Example 2 illustrates a -picture- for a four octal digit field with a sign. This field therefore occupies 5 spaces.

Example 3:

0000

Example 3 illustrates a -picture- of a four digit decimal field plus sign.

Example 4:

XXXX

Example 4 illustrates a four character variable alphanumeric field. A literal or the address of a variable alphanumeric area is a valid input parameter for this picture.

Example 5:

00.000

Example 5 illustrates a floating point picture for a fractional conversion without an exponent part.

Example 6:

00.000 × 000

Example 6 illustrates a floating point picture for an exponent conversion. This type of picture is the most general floating point conversion and should always be used whenever any doubt about the magnitude of output parameters exists.

LITERAL FIELDS:

Example 7:

<TITLE 1>

Example 7 illustrates a literal field in which the alphanumeric -TITLE 1- will be displayed.

SPACING AND LINE CONTROL:

Example 8:

|10|

Example 8 illustrates the technique by which one can space pictures or literal fields. The number between the absolute signs, which designates the number of spaces desired, is always decimal and may be zero.

Example 9:

/

Example 9 illustrates a carriage return or -proceed to the beginning of the next line- symbol.

Example 10:

↑

Example 10 illustrates the top of form operator.

Example 11:

↑↑

Example 11 illustrates the terminate function, or complete dump, operators.

INSERT, DELETE AND OVERPUNCH:

Any of the magnitude symbols (decimal point, multiplication sign, etc.) may be deleted with the following notation:

Example 12:

00}.{00}×{00

Example 12 illustrates a floating point conversion in which the decimal point and multiplication sign have been suppressed using only 8 columns (6 digits and 2 signs).

Example 13:

→00}.{00}×{→00

Example 13 is basically the same as Example 12 with both signs overpunched on the first digits of their corresponding fields. Example 13 output requires only 6 columns.

Example 14:

00{alpha-numeric}00

Example 14 shows an -insert- of an alpha-numeric field in a number which is converted as a decimal integer.

Inserts may be made in any -picture-, but no more than 5 inserts are allowed in a single picture.

REPEAT FORMATS:

Example 15:

(10:000)

Example 15 illustrates a condensed notation for 10 decimal parameters.

Example 16:

(3: XX |3| <TITLE> 00.00×0 ↑)

Example 16 illustrates the universality of the repeat operation. Any picture, literal field or spacing or line or

paper control operator may be included in the scope of the repeat operation.

Example 17:

```
(2: (3: 00 |2|) < Z = > 00 ) /
```

Example 17 shows the use of nested repeats. Up to three repeats may be nested.

TABLE I: NELIAC SYMBOLS

CHARACTER FLEX and CARD	INTERNAL OCTAL CODE	CHARACTER FLEX	CHARACTER CARD	INTERNAL OCTAL CODE
space	00	5		40
A	01	6		41
B	02	7		42
C	03	8		43
D	04	9		44
E	05	8	OCT	45
F	06			46
G	07	;	$	47
H	10	:	.	50
I	11	:	"	51
J	12	(	(	52
K	13	)	)	53
L	14	[	($	54
M	15	]	$)	55
N	16	~	BEGIN	56
O	17	~	END	57
P	20	=	EQ	60
Q	21	≠	NQ	61
R	22	≥	GQ	62
S	23	<	LS	63
T	24	≤	LQ	64
U	25	>	GR	65
V	26	→	=)	66
W	27	+	+	67
X	30	-	-	70
Y	31	/	/	71
Z	32	×	*	72
0	33	not used		73
1	34	|	'	74
2	35	∪	OR	75
3	36	∩	AND	76
4	37	↑	**	77

NOTE: Alphabetic operators must be preceded and followed by a blank column on the card.

TABLE I

Table II: NELIAC CO/NO TABLE

	,	;	.	:	(	)	[	]	|	|	=	≠	≥	<	≤	>	→	+	-	/	×	⍴	|	∪	∩
,	4	4	3	29	6	4	23	25	0	4	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
;	4	4	3	29	6	4	23	0	21	4	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
.	4	4	3	29	6	4	23	0	21	4	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
:	4	4	3	5	6	4	23	0	21	4	10	10	10	10	10	10	10	11	11	17	13	0	0	27	27
(	0	0	0	0	6	20	23	0	0	0	0	0	0	0	0	0	0	20	7	7	17	13	0	0	0	0
)	0	0	0	9	0	8	23	0	21	0	9	9	9	9	9	9	9	8	8	18	14	0	0	9	9
[	24	0	0	0	0	0	0	25	0	0	0	0	22	22	0	22	0	24	24	0	0	0	0	0	0
]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
|	4	4	3	29	6	4	23	25	0	4	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
|	4	4	3	29	6	4	23	25	0	4	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
=	0	0	0	1	1	0	23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
≠	0	0	0	1	1	0	23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
≥	0	0	0	1	1	0	23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
<	0	0	0	1	1	0	23	0	0	0	0	0	0	28	0	0	0	1	1	1	1	1	0	0	1	1
≤	0	0	0	1	1	0	23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
>	0	0	0	1	1	0	23	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
→	19	19	19	19	26	19	23	0	0	19	19	19	19	19	19	19	19	19	19	19	19	19	0	0	19	19
+	0	0	0	27	6	27	23	25	21	0	10	10	10	10	10	10	10	11	11	17	13	0	0	27	27
-	0	0	0	12	6	12	23	25	21	0	12	12	12	12	12	12	12	12	12	17	13	0	0	12	12
/	0	0	0	16	6	16	23	0	0	0	16	16	16	16	16	16	16	16	16	16	16	0	0	16	16
×	0	0	0	15	6	15	23	0	0	0	15	15	15	15	15	15	15	15	15	15	15	0	0	15	15
⍴	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
|	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
∪	0	0	0	0	6	0	23	0	0	0	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0
∩	0	0	0	0	6	0	23	0	0	0	10	10	10	10	10	10	10	11	11	17	13	0	0	0	0

0. FAULT
1. INITIATE RELATION CONTROL
2. FAULT
3. GENERATE STRAIGHT JUMP
4. GENERATE RETURN JUMP
5. CHECK PARTIAL WORD
6. CHECK FOR ALGEBRA
7. CHECK FOR NEG LOOP INCREMENT
8. CHECK FOR LOOP LIMITS
9. CLEAR TEMP LIST
10. GENERATE ADD OR ENTER
11. GENERATE ADD
12. GENERATE SUBTRACT
13. GENERATE MULTIPLY
14. GENERATE MULT QUANT
15. GENERATE MULT OR ENTER
16. GENERATE DIVIDE
17. GENERATE DIV OR ENTER
18. GENERATE DIV QUANT
19. GENERATE STORE
20. INITIATE LOOP CONTROL
21. SET EXIT CONDITIONS
22. GENERATE IO
23. INITIATE SUBSCRIPT
24. MODIFY SUBSCRIPT
25. SET SUBSCRIPT
26. SAVE CURRENT OPERATOR
27. GENERATE ADD OR ENTER
28. INITIATE RELATION CONTROL
29. GENERATOR EXIT.

This table is included as a guide to the legal CO/NO pairs. The numbers given at the intersections specify which generator routine manufactures the machine code instructions pertinent to that pair. In general, if no number is given, that CO/NO pair is illegal. Some special cases, such as shifts or octal notation, are processed elsewhere and do not appear at all.

TABLE II

APPENDIX A

DEFINITION OF NELIAC SYMBOLS

alphabet, a through h, and q through z	- Operands, i.e. constants, variables, names or tags.
alphabet, i through n	- Indices, register variables. These are the AN/USQ-20 B-registers 1 through 6 respectively.
numerals	- Constants. They are always considered to be decimal unless indicated otherwise, see s.
,	- Punctuation. Separates statements in the flowchart logic, names in the dimensioning statement: indicates return transfer operations.
.	- Puncuation. Indicates end of an algorithm, an unconditional transfer and may indicate end of a true or false alternative.
;	- Punctuation. Separates the dimensioning statement from the flowchart logic. Marks the end of an alternative. Separates the input and output parameters in the function definition or call.
:	- Punctuation. When used following a relationship symbol it marks the end of a comparison. It also denotes that which follows as the definition of the name immediately preceeding. In the dimensioning statement it is used when defining partial words or congruent tables.

( ) - Grouping symbols. In the dimensioning statement parentheses enclose the number of locations to be set aside for the name preceeding them. In the flowchart logic they enclose the numerals which indicate the increment or decrement to be added to the index which controls a loop, or the bit limits in a partial word operation. Parentheses also indicate algebraic groups which are to be treated as a whole. They also enclose comment statements.

[ ] - Grouping symbols. These subscripting symbols are used to enclose the subscript operand.

{ } - Punctuation. The braces enclose a subroutine, a function, or a comparison considered as one of the alternatives of a previous comparison. They are also used in dimensioning partial words, jump tables and address switches.

= ≠ < ≧ > ≦ - Relationship symbols. These are used in formulating a decision, branch point or comparison. The < and > are also used as quotation marks in literals and declarations and to indicate input or output in active I/O statements.

→ - Store symbol. That which preceeds it is to be stored in the variable which follows it. It is also used to indicate the limits of bit fields in partial word dimensioning.

×2↑ ×2↑ × / + - Arithmetic symbols. The result of a computation stays in an arithmetic register and is not preserved unless a store operation is indicated. The symbols are listed in priority of execution within algebraic grouping.

∪ - Punctuation. Boolean comparison -or-.

∩ - Punctuation. Boolean comparison -and-.

s - Octal symbol. Indicates that the numerals which preceded are an octal number. When used for machine code the first five digits which preceded the symbol are machine code for the f, j, k and b designators of a command. The next digits to follow are decimal unless indicated otherwise and are inserted as the y part of the instruction; a name may be inserted instead of numerals.

×2↑
/2↑ - Exponent Co/No combination. Scaling technique which indicates that the number preceding is to be shifted (to the left with the multiply symbol and to the right with the divide symbol) the number of bits indicated by the number following. Scaling is not legal with floating point.

| - Absolute sign. When inserted in a name definition the name will be temporary.

APPENDIX B

GLOSSARY OF NELIAC TERMS

The following terms and definitions may be all or in part well known to the reader. However, several have a more or less special meaning in the explanation of NELIAC, so all should be reviewed and understood.

The first explanation or definition will be from -Glossary of Terms and Expressions in the field of Computers and Automation- published in Computers and Automation, Dec. 1954, Vol. 3, No. 10, with a few modifications. The second will be the NELIAC definition if appreciably different. Either the first or second definition will be omitted if not applicable.

ADDRESS VARIABLES
2. A noun which specifies the address of the variables which contain the data or the address of the data itself.

BITS
1. A binary digit; the smallest unit of information; a -yes- or -no-; a single pulse in a group of pulses.

CO
1.2. Current operator.

COMPARISON
1.2. The act of comparing and, usually, acting on the result of the comparison. The common forms are comparison of two numbers for identity, comparison of two numbers for relative magnitude, and comparison of the two signs, plus or minus.

COMPARISON STATEMENT
1.2. A NELIAC statement which designates the type of comparison to be made and the action to be taken as a result of the comparison.

COMPILER
1. A program making routine which produces a specific program for a particular problem by the following process: (a) determining the intended meaning of an element of information expressed in pseudo-code; (b) selecting or generating (i.e. calculating from parameters and skeleton instructions) the required subroutine; (c) transforming the subroutine into specific coding for the specific problem, assigning specific memory registers, etc., and entering it as an element of the problem program; (d) maintaining a record of the subroutines used and their position in the problem program; and (e) continuing to the next element in pseudo-code.

COMPUTER

1. A machine which is able to calculate or compute, that is, which will perform sequences of reasonable operations with information, mainly arithmetical and logical operations. More generally, it is any device which is capable of accepting information, applying definite reasonable processes to the information, and supplying the results of these processes.

CONSTANTS

1.2. A specific numeric value, which is octal 77777 77777 or less, that is in the NELIAC flowchart logic (i.e. DOLLARS × 100 → CENTS, where 100 is the constant).

CONTROL

1. To direct the sequence of execution of the instructions to a computer.

CONTROL ROUTINE

1.2. A routine which is entered with a straight jump and effects control.

DEBUG

1. To isolate and remove malfunctions from a computer or mistakes from a program.

DECLARATION

2. A machine dependent operation called by an active statement in the flowchart logic and inserted into the object program as an open subroutine.

DECLARATOR

2. Any of the names available as English phrases to describe and call on input/output declarations for the AN/USQ-20 computer.

DECREMENT

1.2. To decrease the value contained in a register or cell by a given amount.

DIMENSIONING STATEMENT

1.2. The initial portion of a flowchart which contains the assigned names (nouns) of all variables, lists, and tables used in the flowchart logic.

EQUALITY SIGN

1.2. The symbol (=) meaning -equal to-.

FLEXOWRITER

1.2. A typewriter-like machine which will produce a punched paper tape that can be read by the computer.

FLOATING POINT
1.2. A mode of arithmetic in which each variable has an associated radix point which is adjusted to preserve the maximum precision in each arithmetic operation, independent of the original magnitudes of the variables.

FLOWCHART
1. A graphical representation of a sequence of programming operations, using symbols to represent operations such as compute, substitute, compare, jump, copy, read, write, etc.

2. The dimensioning statement and the flowchart logic.

FLOWCHART LOGIC
1.2. The NELIAC language logic flow using NELIAC operator symbols, constants, predefined variables (nouns), and other routine, control routine and subroutine names (verbs).

INCREMENT
1.2. To increase the value contained in a register or memory cell by a given amount.

INDEXING
1.2. Modifying or altering the operand by an indicated amount or value contained in a register.

INSTRUCTION
1. A machine word or set of characters in machine language which directs the computer to take a certain action. More precisely, a set of characters which defines an operation together with one or more addresses (or no address) and which, as a unit, causes the computer to operate accordingly on the indicated quantities.

JUMP
1. An instruction or signal which, conditionally or unconditionally, specifies the location of the next instruction and directs the computer to that instruction. A jump is used to alter the normal sequence in the control of the computer. Under certain special conditions, a jump may be caused by the operator throwing a switch.

K-DESIGNATOR
1.2. The portion of an AN/USQ-20 machine instruction which designates what is to become the operand of the instruction.

LOAD FLOWCHARTS
1.2. The load program portion of the NELIAC compiler.

LANGUAGE
1.2. A system of communication in which given combinations of given symbols communicate a specific meaning.

LOAD PROGRAM
1.2. A short preliminary program loaded in memory which permits some interpretation and editing of the data during the loading operation.

LOOP
1.2. A loop is a series of operations repeated any number of times as specified by the loop control, or until an exit condition is satisfied.

LOOP CONTROL
1.2. The part of the loop which specifies the number of times the loop shall be repeated.

MACHINE NEGATIVE NUMBER
1.2. Any negative number in the machine kept in complemented form with a one bit in the highest order position.

NO
1.2. Next operator.

OPEN SUBROUTINE
1.2. A sequence of instructions which are built into the program every time they are needed; as contrasted with a closed subroutine, where the instructions are inserted only once, then called with a return transfer instruction.

OPERAND
1.2. Any one of the quantities entering into or arising from an operation. An operand may be an argument, a result, a parameter, or an indication of the location of the next instruction.

OPERATOR
1.2. The person who actually operates the computer, puts problems on, presses the start button, etc.
2. The punctuation and algebraic symbols which the compiler uses to generate machine code.

PSEUDO-CODE
1.2. An arbitrary code, independent of the hardware of a computer, which must be translated into computer code if it is to direct the computer.

PROGRAM

1. A precise sequence of coded instructions for a digital computer to solve a problem.

2. A collection of flowcharts with their associated dimensioning statements from which the compiler manufactures a machine coded program.

PARAMETER

1.2. In a subroutine, a quantity which may be given different values when the subroutine is used in different parts of one main routine, but which usually remains unchanged throughout any one such use.

REGISTER

1.2. The hardware for storing one machine word.

REGISTER VARIABLES

2. The index registers B1 through B6, as represented by the letters i through n, on the AN/USQ-20 computer.

ROUTINE

1.2. See -program-.

SHIFT

1.2. To move the character of a unit of information columnwise right or left. In the case of a number, this is equivalent to multiplying or dividing by a power of the base of notation (usually ten or two). This is regularly performed faster than usual multiplication or division.

STOP CODE

1.2. On punched paper tape, a signal to stop equipment while reading or duplicating a tape.

SUBROUTINE

1. A short or repeated sequence of instructions for a computer to solve part of a problem: a part of a routine.

2. The sequence of instructions necessary to direct the computer to carry out a well-defined mathematical or logical operation: a sub-unit of a routine.

TAPE

1.2. Any kind of paper, metal, plastic, magnetic or non-magnetic material which carries coded information as polarized magnetic spots or punched holes in the tape.

VARIABLES

1. Any specified memory cell or register may be thought of as a variable.

2. Variables in NELIAC are designated by alpha-numeric names. The names must begin with a letter and may be of any length. The compiler will, however, interpret only the first fifteen characters of the name. A variable may also have a specified constant value throughout the program.

WORD

1.2. An ordered set of characters which has at least one meaning, and is stored and transferred by the computer circuits as a unit. Ordinarily, a word has a fixed number of characters, and is treated as an instruction by the control unit and as a quantity by the arithmetic unit.

APPENDIX C

SYSTEM DECLARATION

The active input/output statements listed here call on the -System Declaration-, which is a part of NELIAC. Following the name of each statement is a valid example of the use of that statement.

There are four legal types of operands used in active input/output statements:

1) Address variables,
2) Read operands (register variables, whole or half words),
3) Store operands (whole or half words only),
4) Buffer operands (described in Section IV).

The examples listed below use descriptive names as operands. -Entry- is defined as an entry point, -find index- is a store operand, -core area- is a buffer operand, and all the other operands are read operands.

ACTIVE INPUT/OUTPUT STATEMENTS

```
STOP =
     [stop < entry >,],

START FLEX =
     [start flex < ,],

TURN OFF FLEX =
     [ turn off flex < , ],

READ MAG DRUM =
     [ read mag drum < drum operand >, < core area >, ],

WRITE MAG DRUM =
     [ write mag drum < drum operand>, < core area>, ],

MOVE BLOCK =
     [ move block < start operand >, < number of cells >,
     < move operand >, ],

PRINT LITERAL =
     [ print literal < address variable >, ],

START READER =
     [ start reader < , ],
```

```
TURN OFF READER =
     [ turn off reader < , ],

START PUNCH =
     [ start punch < , ],

TURN OFF PUNCH =
     [ turn off punch < , ],

OUTPUT FLEXCODE =
     [ output flexcode < read operand > , ],

KEY 1 =
     [ key 1 < entry > , ],

KEY 2 =
     [ key 2 < entry > , ],

KEY 3 =
     [ key 3 < entry >, ],

KEY 5 =
     [ key 5 < entry > ,],

KEY 6 =
     [key 6 < entry > ,],
```

```
KEY 7 =
     [ key 7 < entry > ,],

READ ONE FRAME =
     [ read one frame < store operand > ,],

STORE REMAINDER =
     [ store remainder < store operand > ,],

RETURN JUMP STOP =
     [ return jump stop < entry > ,],

SEARCH ZERO =
     [ search zero < list length to search > ,
     < name of list > , < no find entry > ,
     < find index > ,],

SEARCH NOT ZERO =
     [ search not zero < list length to search > ,
     < name of list > ,  no find entry > ,
     < find index > ,],

SEARCH LESS THAN =
     [ search less than < argument > ,
     < list length to search > , < name of list > ,
     < no find entry > , < find index > ,],
```

```
SEARCH GREATER THAN =
     [ search greater than < argument > ,
     < list length to search > , < name of list > ,
     < no find entry > , < find index > ,],

SEARCH EQUAL =
     [ search equal < argument > , < list length to search > ,
     < name of list > , < no find entry > , < find index > ,],

SEARCH BETWEEN =
     [ search between < lower argument > , < upper argument > ,
     < list length to search > , < name of list > ,
     < no find entry > , < find index > ,],

SEARCH NOT BETWEEN =
     [ search not between < lower argument > ,
     < upper argument > , < list length to search > ,
     < name of list > , < no find entry > , < find index > ,],

CLEAR CELLS =
     [ clear cells < number of cells to clear > ,
     < start operand > ,],
```

STOP = < machine($61400_8$0k) >,

START READER = (4) (external function (40_8)),

TURN OFF READER = (4) (external function (400_8)),

START PUNCH =(4) (external function (20_8)),

TURN OFF PUNCH = (4) (external function (200_8)),

KEY 1 = < machine($61100_8$0k) > ,

KEY 2 = < machine($61200_8$0k) > ,

KEY 3 = < machine($61300_8$0k) > ,

KEY 5 = < machine($61500_8$0k) > ,

KEY 6 = < machine($61600_8$0k) > ,

KEY 7 = < machine($61700_8$0k) > ,

READ ONE FRAME = < buffer >, (jump active),

STORE REMAINDER = < machine($15000_8$0k) >,

RETURN JUMP STOP = < machine($65400_8$0k) >,

SEARCH ZERO = < machine($70230_8$0k) > ,
< machine($11437_8$77776$_8$k) > , < machine($61000_8$0k) > ,
< machine($16730_8$0k) > ,

SEARCH NOT ZERO = < machine($70230_8$0k) > ,
< machine($11537_8$77776$_8$k) > , < machine($61000_8$0k) > ,
< machine($16730_8$0k) > ,

SEARCH LESS THAN = < machine($10030_8$0k) > ,
(machine($27000_8$1)), < machine($70230_8$0k) > ,
< machine($04237_8$77776$_8$k) > , <machine($61000_8$0k) > ,
< machine($16730_8$0k) > ,

SEARCH GREATER THAN = < machine($10030_8$0k) > ,
< machine($70230_8$0k) > , < machine($04337_8$77776$_8$k) > ,
< machine($61000_8$0k) > , < machine($16730_8$0k) > ,

SEARCH EQUAL = (machine($10040_8$77777$_8$)), < machine($11030_8$0k) > ,
< machine($70230_8$0k) > , < machine($43437_8$77776$_8$k) > ,
< machine($61000_8$0k) > , < machine($16730_8$0k) > ,

SEARCH BETWEEN = < machine($11030_8$0k) > , < machine($10030_8$0k) > ,
(machine($27000_8$1)), < machine($70230_8$0k) > ,
< machine($04427_8$77776$_8$k) > , < machine($61000_8$0k) > ,
< machine($16730_8$0k) > ,

SEARCH NOT BETWEEN = < machine($11030_8$0k) > ,
< machine($10030_8$0k) > , (machine($21000_8$1)),
< machine($70230_8$0k) > , < machine($04537_8$77776$_8$k) > ,
< machine($61000_8$0k) > , < machine($16730_8$0k) > ,

CLEAR CELLS = < machine($70100_8$0k) >, < machine($16030_8$0k) >,