NRL Report 5976
NAREC Reference 29

NELIAC-N

A TUTORIAL REPORT

J. W. Kallander

Applied Mathematics Staff
Office of Director of Research
U.S. Naval Research Laboratory

and

R. M. Thatcher

U.S. Naval Post Graduate School
Monterey, California

June 17, 1963

U. S. NAVAL RESEARCH LABORATORY
Washington, D.C.

PREFACE

The purpose of this document is to describe in detail the syntax of the U. S. Naval Research Laboratory NAREC version of the NELIAC language; namely, NELIAC-N. This version of the NELIAC compiler was written by Charles A. Tapella of the U. S. Navy Electronics Laboratory, San Diego, California, and John W. Kallander of NRL, was obtained through the courtesy of Dr. Maurice H. Halstead, Head, Computing Center, NEL, and was implemented on the NAREC by John W. Kallander. NELIAC-N is based on and is very similar to NELIAC-T-1604.

This document is tutorial in nature and is not intended to be definitive of NELIAC-N. The report "The NELIAC Compiler Language, U. S. Naval Postgraduate School CDC-1604 Version", was written by Richard M. Thatcher, Department of Operations Research, USNPGS, Monterey, California, and published by the USNPGS in January 1963. This CDC-1604 Version Report has been rewritten to pertain to NELIAC-N and expanded by John W. Kallander of the Research Computation Center, NRL, and the result is this document. An additional report defining NELIAC-N will be issued at a later date.

However, this tutorial report should be studied in detail by any person considering programming in NELIAC-N, and should be thoroughly understood before using the definitive report which will follow.

Dr. Halstead's published book Machine-Independent Computer Programming (Spartan Books, Washington, D.C., 1962) describes the basic NELIAC language, provides guidance in developing compiler programs and contains much interesting background regarding NELIAC that could not be included in this description of the NELIAC language as implemented on a particular computer. It is desirable, although not necessary, that the user of this document read through the first three chapters of Dr. Halstead's book before, or concurrently with, studying this more detailed work.

Credit is due Sidney W. and Catherine B. Porter, Computing Center, NEL, for writing NELIAC 1604-N, the intermediate compiler used to debug NELIAC-N to the point of self-compilation; to Maurice Brinkman, RCC, NRL, for his considerable and prolonged aid while debugging the compiler and training NRL's programmers and scientists in the use of NELIAC-N; and to Mrs. Elizabeth Wald, also of the RCC, for writing the NELIAC-N Library of Functions.

Much credit also must be extended to Mrs. Rose Skinner, Branch Secretary, RCC, for typing and correcting the compiler flowcharts through all of its numerous recompilations, for typing the extensive group of test programs necessary to raising the NELIAC-N compiler to its present level of development, and for typing this entire manuscript.

Richard M. Thatcher
Dept. of Operations Research
U. S. Naval Postgraduate School

John W. Kallander
Research Computation Center
U. S. Naval Research Laboratory
April 1963

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

ABSTRACT

This report contains a tutorial description of NELIAC-N, the version of the NELIAC language implemented on the NAREC by means of the NELIAC-N compiler. NELIAC is a problem-oriented, machine-independent programming language which enables programmers, scientists, and engineers to write their programs in a mathematical language rather than requiring an actual machine language or an assembly language. NELIAC thus minimizes the knowledge of the actual computer required by the programmer, maximizes the readability of the programs themselves, and provides carry-over value of programs from one computer to another.

PROBLEM STATUS

This is an interim report; work on this problem is continuing.

AUTHORIZATION

NRL Problem B02-03B
Project RR 003-09-41-5101

Manuscript submitted May 31, 1963.

NAREC REFERENCE #29

NELIAC-N, A TUTORIAL REPORT

I. INTRODUCTION

A NELIAC program is a means of expressing a computer problem in terms much closer to an algebraic language than the detailed step-by-step instructions of actual machine language. A program written in the NELIAC language is comprised of statements and proper punctuation. This language is interpreted and translated by the NELIAC compiler which generates the actual machine instructions or object program understood by a computer. One must, therefore, adhere strictly to the rules of the language as each statement, set-off by proper punctuation, has definite significance to the compiler.

CHARACTERS OF THE NELIAC LANGUAGE

The NELIAC vocabulary is constructed from the following symbols:

THE NELIAC CHARACTER SET

1 2 3 4 5 6 7 8 9 0

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

, ; : .

( ) [ ] { }

+ - * / ↑ → |

= ≠ < > ≤ ≥

∪ ∩ #

Although the uses of the characters are described in detail later in this document, it might be well to note here the names of the last 26 of them:

, Comma

; Semicolon

: Colon

. Period

() Left and right parentheses

[] Left and right brackets

{} Left and right braces

+ Plus

\- Minus

* Multiply

/ Divide

↑ Exponent sign, or Up arrow

→ Arrow, or Right arrow

| Absolute sign

= Equal

≠ Not equal

< Less than

> Greater than

⩽ Less than or equal to

⩾ Greater than or equal to

∪ Or

∩ And

Hexi sign

Statements, each denoting a specific action, are built from this character set into a NELIAC program.

GENERAL PROGRAMMING RULES

All computer programs require part of the computer memory for storage of numerical values pertinent to the problem. These memory locations are used by the program in the sense that the program obtains values from them in order to perform indicated operations on them. These memory locations are set by the program in the sense that the program stores intermediate and final results of computation into them. Thus, any program can be broken into two parts: the storage part and the operating, or program logic, part.

When a programmer writes a program in compiler language he must tell the compiler what the storage requirements will be. The compiler automatically handles the problem of deciding which locations of memory will actually be used for storage. In the NELIAC language, storage requirements are specified by the programmer by making up identifiers or names to which the compiler program will automatically assign memory locations. Throughout a given program, any name, once assigned, will refer to the same memory location or group of memory locations. An exception to this rule (namely, temporary or local names) will be explained later. The numerical values contained by these memory locations are then referenced by name in the program logic part where dynamic operations are indicated. Consider the following example:

Algebraic Equation	NELIAC Statement
A + B = C	, A + B → C ,

The algebraic equation states that the value of A is added to the value of B. This sum is equivalent to the value of C. The NELIAC statement is more dynamic in that a certain action is implied by the right arrow. This right arrow is a store operator; thus, the value in the memory location referenced by the name A is added to the value referenced by the name B and the sum is stored into the memory location named C. That the store operator is not equivalent to the equal sign can be seen from the following example:

, A2 + 1 → A2 ,

The NELIAC statement says to add one to the value in the location referenced by the name A2. This sum is to be computed and stored back into the location referenced by A2 thereby replacing the old value by the new.

NELIAC PROGRAM STRUCTURE (General)

The two parts of a computer program, the storage part and the operating part, are handled in NELIAC by the dimensioning statement (or noun list) and the program logic (or body of the program), respectively.

In the dimensioning statement, the programmer specifies storage requirements by making up names to which the compiler will assign storage locations. Each location so named is called a variable since it is possible for the program to change its value. A group of memory locations to which the programmer assigns only one name is defined as a table (of variables), also called an array (a one-dimensional array usually being referred to as a list). Later in this document it will be seen how the programmer may assign a name to part (i.e., certain bits) of a memory location or in the case of a table (array or list), how he may assign a name to the same part of each location of the table. Each part-memory location so named then becomes a variable. In the dimensioning statement the programmer also assigns initial values and specifies the mode and number format of each variable and indicates output formats for variables whose values are to be printed.

The program logic is the operating part of the program which indicates the sequence of dynamic operations to be performed. Basic to the structure of the program logic are the statements of which it is comprised. Comparable to ordinary English, statements of program logic are set off by punctuation symbols of which there are 5:

- , Comma
- ; Semi-colon
- : Colon
- . Period
- .. Double Period

the double period being used only to indicate the end of the program logic part and, hence, the end of the flowchart (or subprogram). Following is an example of two statements which might be used to compute the expression

$$\frac{A \times B + C}{A \times B - 2C}$$

and store the result into location G:

```
, A * B → H , (H + C) / (H - 2 * C) → G,
```

This is not a complete program, however. Only part of the program logic is illustrated above. Every name used by these statements must be defined beforehand or later in a dimensioning statement (or in a function definition). A complete flowchart to perform this simple task for specified values of A, B, and C might be as follows:

NELIAC FLOWCHART	NOTES
5	Load Number signifying the beginning of the flowchart to the compiler.
A = 1, B = 2, C = 1, H, G,	Dimensioning Statement: Initial values are specified and names assigned to each memory location. Note that locations are allocated and given an initial value of zero when initial values are not speci-field. A final comma in the dimensioning statement is normally omitted since the semicolon also functions as this comma.
;	The first semicolon indicates the beginning of the operational portion of the flowchart.
COMPUTE:	COMPUTE is the name of this flowchart. This type of statement is called a definition or label.
A * B → H, (H+C)/(H-2*C)→G,	Program logic: A strict left to right flow is followed. Spacings, indentations, blank lines do not alter the flowchart in any way (except in the case of the ALGOL words which will be explained later). A final comma in the program logic is normally omitted since the double period also functions as this comma (except for subroutine and function calls).
..	The double period indicates the end of the flowchart.

NELIAC FLOWCHART

Although a NELIAC program may consist of a single dimensioning statement followed by a single block of program logic and, indeed, short NELIAC programs are written in this form, it is very convenient and, at times, absolutely necessary, to be able to write programs as a series of subprograms called flowcharts, each of these flowcharts having the form of a NELIAC program; i.e., a dimensioning statement followed by the program logic. All of the subprograms or flowcharts comprising a single NELIAC program are compiled together in a single compiler sweep in an order determined by the programmer just as if the entire program were written as a single unit. Hence, a programmer may write and check out a long program as several independent units; in fact, the flowchart concept makes feasible the compilation of long and difficult programs whose various subprograms have been written and checked out by different programmers. In addition, the flowchart concept makes the correction of program units, the substitution of new units for old units, and even the addition and removal of units, a trivial procedure. Finally, the finite memory space of any computer requires that very long NELIAC programs (more than ten to fifteen double-spaced typed pages in the case of the NAREC) be written as two or more separate flowcharts; although, even here, the number, size, and arrangement of the flowcharts is still entirely up to the programmer's discretion subject solely to the limitation that no flowchart exceeds the maximum length dictated by a computer memory size.

Inasmuch as the structure of and the language used in each of these subprograms are identical to the structure and language of a program written as a single NELIAC unit (or flowchart), the programmer need only consider a program as consisting of a single unit throughout most of this document. Toward the end of the document, he will see how the extension of everything he has learned about the NELIAC language and the NELIAC program naturally applies to multiple-unit programs.

COMMENTS

It is often helpful to insert comments in English to the NELIAC language in order to clarify the meaning of the program to the reader. This capability is provided by NELIAC-N according to the following rules:

1. Enclose the comment in parentheses.

2. A colon must be placed as the next operator after the left parenthesis. The colon may be placed immediately after the parenthesis, or any word or phrase which meets the NELIAC definition of a name may be inserted between them. The word COMMENT is customarily inserted here.
3. Any words, numbers, or symbols may be included in the comment with the exception of the right parenthesis which signals the end of the comment and the double period (..) which signals the end of the flowchart to the compiler.
4. Comments may be inserted between any two statements of the dimensioning statement or the program logic.
5. Normal punctuation should either precede or follow the parentheses.

EXAMPLE:

```
, A → B, (COMMENT: A → B means to store the
          current value of location A into
          location B.)
```

Of course, comments are meant to be an aid only to the reader of the program and have no meaning whatsoever to the compiler.

ALGOL WORDS

In addition to the ALGOL word COMMENT, whose use has been described in the preceding section, NELIAC also provides, in a slightly different sense, for the use of the ALGOL words

GO TO

DO

IF

IF NOT,

and, FOR

to describe (but not define or specify as in ALGOL) certain procedures in the flowchart. These five words (or word phrases) when written as above; i.e., when set off by spacing except IF NOT, which must be immediately followed by a comma (which may or may not be preceded by spacing), and with internal spacing in GO TO and IF NOT, are known, in NELIAC, as ALGOL words and have special significance in the flowchart. They are parenthetical to the compiler; i.e., they are completely ignored by the compiler (except when inserted within a double period). As such, they may be used to describe certain procedures in the printed copy of the flowchart. However, just as it is certain operator combinations which determine (or define) a comment, the word COMMENT having no meaning (if used at all), it is certain operator combinations, and only these operator combinations, which determine these procedures, the descriptive ALGOL words having no meaning (if used at all) to the compiler. The sole function of these words is to improve the readability of the printed copy of the flowchart. In fact, the compiler will completely ignore these words no matter where they are used in the program (except within a double period). The use of the individual parenthetical words will be described as the procedures to which they apply are defined.

However, if any of these character combinations are used without the spacing (multiple spaces being equivalent to a single space) described above in their definitions, the character sequence will be considered, not as an ALGOL word to be ignored, but as a bona fide part of the program. Hence, these character combinations may be used as portions of names defined by the programmer. It should be borne in mind that spacings, indentations, and blank lines may alter a NELIAC program only in the possible determination of these ALGOL words.

II. THE STORAGE PART

DEFINITION OF NAMES

Names are the means by which the programmer refers to and manipulates the quantities in which he is interested in NELIAC programs. In particular, each name defined by the programmer is assigned a cell or location in the computer memory (or part cell in the case of partial words). NELIAC names are divided into two major classes: nouns and verbs. Nouns are those names defined in the dimensioning statement of the flowchart and of the function definitions. Verbs are those names defined in the program logic (excluding the dimensioning statements of function definitions) and, as will be seen later, are actually labels or names of procedures. The rules of formation of all names whether nouns or verbs are the same and will be given here although only the definition and usage of nouns will be discussed. At the time the definition and usage of the various verbs are discussed, it should be borne in mind that the general rules of formation of NELIAC names given here apply to verbs also.

Nouns are the means by which a programmer writing in NELIAC controls the use of computer memory locations for storage. He assigns a name (specifically, a noun) to each single memory location, each group of memory locations, to each part-memory location or to each group of part-memory locations used for storage. The name itself is left to the imagination of the programmer limited only in that it must begin with a letter of the alphabet, must contain only letters, spaces, and numbers, and must be uniquely determined within its first 16 characters excluding spaces and ALGOL words. Capital and lower-case letters are interchangeable and may therefore be used at the discretion of the writer. Single letters, with the exception of I, J, K, L, M, and N, are permissible names. These letters - I, J, K, L, M, and N - when standing alone refer to the six index registers which are always automatically available as fixed-point, full-word integers having four hexadecimal digit IO format and which, consequently, must never be dimensioned (except as temporary names or as dummy parameters in function definitions, both of which will be explained later). Other names used by the compiler will be discussed in the appropriate chapters and are listed in Appendix C.

Examples of legal NELIAC names:

Q
MA 10
INTEGRAL
L2350 HL 543
BEGINNING OF FLOWCHARTS
FORMULA
COMMENT

CONSTANTS AND VARIABLES

A constant is a value not defined by name in the dimensioning statement but written explicitly in the program logic. Note the example:

$$A2 + 1 \rightarrow A2$$

where 1 is the stated constant. A constant is thus distinguished from a variable, the latter being defined in the dimensioning statement and referenced by name throughout the program logic. A variable may or may not actually change its value during the operation of the program.

All numbers in NELIAC may be written in either one of two modes, fixed point integer or floating point format. Floating point numbers differ from fixed point in allowing for decimal fractions as well as integers, and, therefore, much greater accuracy in computation without requiring scaling. These numbers are commonly and easily used in computer problems as the alignment of decimal points during computation is handled automatically.

Following are examples of fixed point constants written within the program logic.

```
, - 10 → A ,
, 25 - D → C ,
, A - 476 → X ,
, B / (-5) → Y ,
```

In expressing a floating point constant within the program logic, a <u>decimal point</u> must distinguish it from a fixed point value. As machine operations on the two modes, fixed point and floating point, are quite dissimilar, care must be taken to avoid mixing modes in arithmetic or store operations. The examples following illustrate the use of legal floating point constants. (Note: The last example is an illegal statement using mixed modes.)

```
, A - 1.068 → C ,
, 1.0 - D → X ,
, 0.0247 → Y ,
, - 25.0 → Z ,
, 1.0 * - 6 → TOLERANCE,

, 5 - 10.0 → Q, (COMMENT: ILLEGAL STATEMENT)
```

The last example, legalized, might read

```
, 5.0 - 10.0 → Q ,
```

For numbers less than one in absolute value, a zero must be written before the decimal point.

The constant zero, whether fixed or floating point, must always be written as 0 in logic.

DIMENSIONING FIXED POINT VARIABLES

The initial values of variables to be used in a program are set in the dimensioning statement, and names are defined by which they may be referenced. Throughout the program logic, variables are treated either as fixed point or floating point numbers according to the method by which they are defined in the dimensioning statement. Once a variable has been dimensioned there is no way whatsoever of changing its mode or format. In particular storing a number or variable into another variable of the opposite mode will place the current representation of this number or variable into the variable but will not change the mode of the latter variable. Hence, it is strictly forbidden. Example A illustrates legal definitions of variables having decimal fixed-point numbers as initial values.

Example A:

```
NR OF SAMPLES = 25 ,
ALPHA = - 12 ,
BETA = 8437 ,
GAMMA ,
```

Any unique name followed by an equals sign and the value of a decimal fixed point number is sufficient for defining a variable of that name with initial value equal to the given number. Each definition must be separated by a comma. If a fixed point variable is to be given an initial value of zero, the name followed by a comma is sufficient. Numbers are treated as positive unless preceded by a minus sign. In fact, in the dimensioning statement, a positive number may not be preceded by the plus sign, but must be unsigned.

When defining a table of variables, the size or length of this table also must be indicated. The number in parentheses immediately following a name indicates the number of entries in the table. Irrespective of the mode associated with the name, this list length must always be an unsigned fixed point integer - either decimal or hexadecimal. After the equals sign the values of the initial entries, separated by commas, are written. Suppose a table is to contain five variables. Then five memory locations of the computer must be allocated. The following example defines such a table of fixed point numbers called TAB X.

TAB X (5) = 5, 45, 8, -3, 8,

As shall be studied in detail later, individual values of the table may be called upon in the program logic through subscripting of a single name, in this case, TAB X. In mathematical notation, a subscript usually is written as a small character below the line; e.g., TAB X_0 to indicate the first entry of the table, in this instance, to reference the location containing the value 5. TAB X_1 would refer to the second entry, (the value 45), etc. In the NELIAC language subscripting is indicated by the use of <u>brackets</u> around the subscript in the following manner: TAB X [0], TAB X [1], TAB X [2], etc. As subscripting in NELIAC begins with <u>zero</u>, not one, TAB X [3] refers to the fourth entry of the table which (above) contains a value of -3. Since the name TAB X without subscript references the first entry of the table, the use of the notation TAB X [0] is redundant, but it is nonetheless legal.

Note, in the following example, that twenty-five locations are allocated for a table named XCOORD, but only five fixed-point initial values of the table are specified.

XCOORD (25) = 10, 5, -8, 3, 2,

The remaining locations of table XCOORD, since initial values are not explicitly specified, will contain zero quantities. The definition of an entire table with initial values of zero is written; e.g., as

PMATRIX (100),

One hundred memory locations are thus reserved for one hundred fixed point integer values which may be computed and stored into these locations during operation of the program.

Zeroes may be dimensioned implicitly in any cell of a table by the proper use of punctuation. In the example below, part of the table initially contains zero quantities. Of course, the zeroes may also be stated explicitly.

```
XMATRIX (9) = 5, 6, 7,
                   , -3, 4,
                   ,   , 2,
```

In NELIAC-N, the range of fixed point integers which may be explicitly represented is from $-(10^{13}-1)$ through $(10^{13}-1)$ inclusive although NELIAC-N will handle integers which arise in calculations up to the range $-(2^{44}-1)$ through $(2^{44}-1)$ inclusive.

DIMENSIONING FLOATING POINT VARIABLES

Initial floating point values are assigned in the dimensioning statement in much the same manner as fixed point values. The essential difference is that floating point numbers are characterized either by a non-leading decimal point in the number and/or by multiplying the number by a power of ten, the ten being only implicitly stated. (See section headed Constants and Variables for examples of the proper floating point notation of constants in the program logic. All forms of floating point numbers given below for dimensioning are valid forms for use in the program logic with the single exception of the form (number without a decimal point) * (exponent).)

For example, the number 500 is written in scientific notation as $5 \cdot 10^2$. In the NELIAC dimensioning statement, this number might be written as 5 * 2. This number may also be written as 50.0 * 1 (implying $50.0 \cdot 10^1$), or as 5000.0 * -1, 5. * 2, 500., 500.0, etc.. Numbers of very small or large magnitudes are thus conveniently written; e.g., the number 0.00005 is written in scientific notation as $5 \cdot 10^{-5}$, in the NELIAC dimensioning statement as 5 * -5, as an alternate form. The following example illustrates proper dimensioning of floating point numbers:

```
HUNDRED = 100 * 0,
PI = 0.31416 * 1,
OMEGA = -4.25 * -3,
ZERO = 0 * 0,
E = 2.7182818,
FIFTEEN = 15.,
```

A table of floating point values is defined in a manner similar to a table of fixed point values: the defining name followed by the number (in fixed point notation) of entries in the table enclosed within parentheses. However, the entries themselves must be written in floating point notation.

```
FLTING TABLE (5) = 5 * 3, 1.23,
                   0.34, 4.2 * 0,
                   10.8 * -1,
```

In the matter of sign, the exponent of a floating point number differs from all numbers in that the suppression of the plus sign is not required; e.g.

```
        FL NUMBER = 5 * 6,
or      FL NUMBER = 5 * +6,
```

A table, initially zero, later to be filled by the program with computed floating point values may be defined in the following manner:

```
        T TAB (25) = 0 * 0,
or
        T TAB (25) = 0.0,
```

Because of this definition any variable referenced in the program logic by the name T TAB and a subscript (which may be implied for T TAB [0]), will be treated as a floating point variable.

Likewise, a period after a name or an array will define the name or array as floating point with initially zero value or values:

```
ZERO.
T TAB (25).
```

In the case where such a definition is the last definition in a dimensioning statement, both the period and semi-colon are required.

The range of floating point numbers in NELIAC-N is from 10^{-231} through 10^{+307} with characteristics of 36 bit significance (10 decimal places).

HEXADECIMAL NOTATION

A number format conveniently used by a programmer in any part of the program is that of hexadecimal notation. Hexadecimal numbers in the computer are handled as fixed point integers and in NELIAC-N are distinguished from decimal fixed point integers by a preceding hexi sign. Hence, one defines hexadecimal numbers in the dimensioning statement as illustrated in the examples below:

```
HEXADECIMAL NR = #2ab7,
MASK 1 = #7f ffff ff,
HEXI TABLE (3) = #26a8,
                 #6754,
                 #ffff,
NEG HEXI NR = -#3A7,
```

Hexadecimal integer constants are entered directly in the program logic and used in arithmetic expressions in exactly the same manner as decimal integer constants:

```
, #7e3 + B → A ,
```

The hexadecimal notation may be used for fixed point integers only, never for floating point numbers. The hexadecimal integers are signed just as other numbers, i.e.; a plus sign must be suppressed, the minus sign immediately precedes the hexi sign.

The range of hexadecimal integers when used as numbers is from $-(2^{44}-1)$, through $(2^{44}-1)$, inclusive. However, NELIAC-N does accept 45 to 48 bit (12 hexa-decimal digit) hexadecimal numbers in the machine-language sense of a NAREC word.

Appendix B is a flowchart illustrating the various forms of dimensioning nouns available in NELIAC-N. The forms illustrated are typical dimensioning entries but are, by no means, exhaustive of the various forms and combinations available.

III. ARITHMETIC OPERATIONS

BASIC OPERATIONS

The basic arithmetic operations in NELIAC are denoted by the following symbols:

- \+ Addition
- \- Subtraction
- * Multiplication
- / Division
- ↑ Exponentiation

A mathematical expression may be built up with any combination of these operators, and algebraic grouping may be as complex as desired. Every series of arithmetic operations should terminate with the storage of the result in either a named variable or an index register by the use of a right arrow or must terminate in a comparison. A NELIAC statement is completed in this manner, and every such statement is terminated by a comma (or its equivalent in special cases). It must be remembered that the mode of the values used in any one expression must be consistent; i.e., fixed and floating point variables and constants may not be mixed. For example, if a variable LOAD has been defined in the dimensioning statement as a floating point variable, then the following statement would be illegal:

,LOAD + 5 → LOAD,

Nor should the result of a fixed point computation be stored into a floating point variable. For example, if the name RESULT is dimensioned as a floating point variable, and the name INTEGER references a fixed point variable, then the following statement would be illegal:

,INTEGER / 5 → RESULT,

The sole exception is the zeroing of a floating point location. If the name RESULT is dimensioned as a floating point variable:

,0 → RESULT,

i.e., the representation of a fixed point zero is used.

In NELIAC-N, a statement may terminate in a sequence of store instructions. In fact, a store instruction need not in itself terminate the series of arithmetic operations since the store instruction and all five of the arithmetic operations listed at the beginning of the chapter are legal immediately after a store instruction. An example is:

,A * B - C → D → E + F → H - I → J * K → L/M → N → O → P → Q,

HIERARCHY OF ARITHMETIC OPERATIONS

The hierarchy of operations consists, first, of exponentiation, second, of multiplications and divisions in sequence from left to right, and, third, additions and subtractions also in sequence from left to right. Parentheses may be used to alter the sequence of operations as needed. The only use for the exponentiation symbol is to multiply or divide a fixed point variable by a positive power of 2. In fact, B * 2 ↑ 5 → B, merely shifts (cycles) arithmetically the contents of B to the left 5 binary places. On the other hand, division by a positive power of 2 arithmetically shifts the variable to the right the indicated number of places.

In NELIAC-N, the notation B * ↑ 5 → B results in the full register (48 bit) shift of the contents of B to the left 5 binary places. The corresponding division notation is used for the full register right shift.

The following examples illustrate hierarchy of arithmetic operations (all statements below are legal):

EXAMPLES OF ARITHMETIC STATEMENTS

NELIAC STATEMENT	EQUIVALENT NELIAC STATEMENT
1) A + B / C → D ,	A + (B / C) → D ,
2) A + B / C + D * E → F,	A + (B / C) + (D * E) → F,
3) A * 2 ↑ 5 / B → Y,	(A * 2 ↑ 5) / B → Y,
4) A / B / C → Z,	(A / B) / C → Z,
5) A / B * C → Z,	(A / B) * C → Z,
6) A - B * C + D → P,	A - (B * C) + D → P,
7) A / B * D / C → P,	((A / B) * D) / C → P,

FIXED AND FLOATING POINT PACKAGES

In NELIAC- N, fixed point multiplication and division is accomplished through return jumps to the subroutines MULTIPLY and DIVIDE respectively, these subroutines being of the fixed point package which is automatically compiled into any program requiring it.

Likewise, floating point addition, subtraction, multiplication, and division are accomplished through return jumps to FLADD, FLSUB, FLMUL, and FLDIV respectively in the floating point package which is automatically compiled into any program requiring it.

Hence, use of these names must be avoided by the programmer since he can never be sure when either or both of these packages will be called into a program containing his flowchart.

IV. TRANSFER OF CONTROL

NORMAL JUMPS

In programming, certain conditions which necessitate skipping over portions of the program to some other point of entry may be met within the program logic. This would necessitate transfer of control of the program to a set of statements other than those continuing in natural sequence. It is necessary, therefore, to label or define that set of statements to which a jump is to be made. This is accomplished by assigning a name (which is thus classed as a verb) preceded by punctuation and followed by a colon to any portion of the program logic.

, ADD: A + B + etc....

A jump to this segment of the program is specified by the use of a period following the definitive name. A statement such as

,ADD.

would immediately transfer control, or jump, to that portion of the program so defined, in this case, A + B + ... The ALGOL word GO TO described in Chapter I may be used for descriptive clarity in the flowchart, in which case the above example becomes

, GO TO ADD.

As is the general case with ALGOL words used in NELIAC, the GO TO is completely parenthetic. The jump is established by the operator combination (punctuation) NAME. .

In the following example, a jump made to MULTIPLY would execute every statement following, including those labelled COMPUTE. The natural sequence of the program is followed unless otherwise specified by a jump statement.

```
, ON:  NR PASSES → CT PASSES,
       MULTIPLY:  A * (B + C) * D → Z,
                  P * Q → Y,

       COMPUTE:  (G * H) / (Y * Z) → ZOO,
```

The assignment of meaningful names to such NELIAC <u>paragraphs</u> often gives greater coherence to a program even though a jump to that name is not specified; this device then becomes merely a labelling device which in itself does not cause generation of machine instructions.

SUBROUTINES AND RETURN JUMPS

In some cases a return jump is desirable; i.e., a jump is made to a special segment of the program called a subroutine. After the subroutine has been executed, control is to be returned to the point of the program logic immediately following that from which the jump was made.

The naming of a subroutine is familiar -- any unique name (which is thus classed as a verb) preceded by punctuation and followed by a colon -- however, the limits of the subroutine must be defined by braces. The subroutine may be as long and complex as desired as long as the limiting braces surround it. Hence, a subroutine is easily recognized by the sequence: punctuation, name, colon, left brace, etc.

Example of a subroutine:

, GENERATE: { RAND, X * Y → Z }

To execute the statements within the braces, the subroutine must be called in the following manner (elsewhere in the program logic):

, GENERATE,

where the definitive name is followed by a comma (except for a subroutine or function call ending an alternative of a comparison, in which case the semicolon ending the comparison customarily replaces the comma), indicating a return jump to the subroutine. The ALGOL word DO may be used here for additional clarity in the printed copy, the word DO, of course, being parenthetic. In this case the preceding example becomes:

, DO GENERATE,

Notice, within the subroutine GENERATE, a call for another subroutine, RAND, is made. After execution of the statements which must be defined by RAND elsewhere in the program, the value of X * Y is stored into the variable Z, and control is transferred back to any statements following the call for GENERATE.

To avoid having the sequence of the main program logic inadvertently flow into a subroutine, all subroutines are customarily written at the end of flowcharts. It is necessary to program jumps around such defined subroutines if they are placed in the way. An example will serve to clarify this point.

```
, A + B → C, CLEAR, NEXT.
CLEAR: { 0 → I → J → K → L → M → N }
NEXT:  C + D → E, etc.
```

In this example, A + B is stored into C, then the 6 index registers I thru N are cleared to zero by calling on the CLEAR subroutine. Then in order to keep the program from illegally trying to operate the CLEAR subroutine as the next sequence of instructions, it is necessary to jump around it to location NEXT, where C + D is stored into E, etc.

It must be noted that while any number of subroutines may be called within another subroutine (except the subroutine itself, of course), no subroutine may be defined within another subroutine.

V. DECISIONS

COMPARISON STATEMENTS

Comparison statements are the means by which questions may be asked regarding relative values of two or more variables or constants. Almost any meaningful question may be asked in the comparison statement by using the following comparison operators:

$$< \quad > \quad = \quad \neq \quad \leqslant \quad \geqslant$$

Basic comparison statements are illustrated below. Note the colon must end the comparison statement.

```
, A < B :
, A > B :
, A = B :
, A ≠ B :
, A ≤ B :
, A ≥ B :
```

These operators may be joined in the general form

```
, A < B ≤ C ≠ D etc. :
```

where the comparison statement has its usual mathematical meaning. This usage will be described in more detail later in this chapter. Immediately following the question (comparison statement) two alternatives are written. The first alternative will be operated if the answer to the question is true; the second, if the answer is false.

COMPARISON STATEMENT	FIRST ALTERNATIVE	SECOND ALTERNATIVE
A = B :	TRUE ;	FALSE ;

An alternative may consist of one or more statements, the last of which is terminated by a semicolon (or a period) rather than a comma to indicate the end of the alternative as well as the end of a statement. Unless an alternative itself breaks up the normal sequence of the program logic by specifying a normal jump to some other part of the program logic, the statement following the false (second) alternative will be operated next. Consider the following examples:

```
, C ≥ D : A * C → E, I + 1 → I ;
  B * C → E ;
  COUNT + 1 → COUNT,
```

Here, a comparison is made: if the value in C is greater than or equal to that of D, then execute the true alternative which stores the value in A times the value in C into location E and adds 1 to index I. If the value of C is less than that of D, execute the false alternative which stores the value in B times the value in C into location E. In either case, continue by executing the statement following the false alternative which adds 1 to COUNT, etc.

In order to make the NELIAC program easier to read, the ALGOL words IF and IF NOT, , parenthetic as always, may be added to the comparison statement complex (See Chapter I). For instance, the last example may be written:

```
, IF C ≥ D : A * C → E , I + 1 → I ;
  IF NOT, B * C → E ;
  COUNT + 1 → COUNT ,
```

These words do not add any meaning to the program, however, and are ignored by the compiler during compilation.

Constants and the index registers of the compiler also may be used on either side of a comparison statement. Again, however, care must be taken to avoid comparing fixed point values with floating point ones. Algebraic grouping may be as complex as desired on the left hand side of a comparison statement, but the right hand side must consist of a single unsigned variable (which may be subscripted and/or bit-handled as explained later) or an unsigned constant. Thus, the following statement is legal:

```
(A + B) / C > D: TRUE; FALSE;
```

while a case such as

```
(COMMENT: ILLEGAL STATEMENT)
D ≤ (A + B) / C:  TRUE; FALSE;
```

is illegal. Note, in the case of comparison statements, the result of an algebraic expression is not necessarily stored into a variable although it may be:

```
(A + B) / C → X > D: TRUE; FALSE;
```

Return jumps and unconditional jumps are legal commands within either alternative. In the case where unconditional jumps are made, the period instead of a semi-colon will end either the true or the false statement. Examples:

```
A > B :  START.  END.
A ≠ B :  C → D, 5.0 + E → F, BEGIN.
         RAND, 1 + J → J, FINISH;
```

Notice how the return jump made to the subroutine FINISH is indicated as FINISH; Though FINISH,; is not in error, the comma would be redundant in this case.

Another illustration of the comparison statement: Suppose it is desired to set Y to one of 3 values according to the following criteria:

```
   8.72    if 0.0 ⩽ X < 10.9
Y=16.19    if 10.9 ⩽ X < 21.6
  24.07    for any other value of X
```

Then, the program is to continue by transferring control to MORE. A NELIAC solution might be:

```
, IF 0 ⩽ X < 10.9 :  ONE, MORE. ;
  IF 10.9 ⩽ X < 21.6 : TWO;  THREE;  MORE.

  ONE : {8.72 → Y }
  TWO : {16.19 → Y }
  THREE : {24.07 → Y }
```

The above solution is by way of illustration. Perhaps a better solution would be:

```
0 ⩽ X < 10.9:
   8.72 → Y;
   10.9 ⩽ Y < 21.6:
      16.19 → Y;
      24.07 → Y;;
MORE.
```

as described in the next section, NESTED DECISIONS.

Note that it is always mandatory to indicate the end of each alternative with either a period or a semicolon once a comparison statement is written.

If nothing is to be done within a single alternative, a semicolon suffices to indicate continuation of the sequence of the program. Example:

A < B > C: ; X → Z; Y → H,

In the case that the relationship in the above example is true, no statements are executed and the sequence of the program continues with the value of Y being stored into H. If any part of the relationship is false, X is stored into Z and the sequence continued with Y being stored into H. The situation may be reversed and nothing done if the relationship is false.

Example:

A < B > C: X → Z;; Y → H,

In all cases, the termination of each alternative must be indicated by either the use of a semi-colon or a period. The number of statements used in either alternative is unrestricted.

NESTED DECISIONS

Decisions may be nested within other decisions. Note the following example:

```
,LOLIMIT < XCOORD:
   RAND, X ≥ MSW PROB:
      5 → MINETABLE;
      - 1 → MINETABLE;;
   NULL → MINETABLE;
```

Begin with the comparison LOLIMIT < XCOORD. If the relationship is true the statements of lines 2, 3, and 4 will be executed; if false, the statement of line 5 will be executed. Within the first true alternative is a return jump to the subroutine RAND and another decision. The true and false alternatives for this second comparison are merely distinguished by semi-colons. With nested decisions, care must be taken to insure that a second comparison is completed within a single alternative of the first comparison.

In order to improve readability in writing comparisons, the convention that successive comparisons will be indented by multiples of three spaces has been adopted. Furthermore true and false alternatives are never placed on the same line (unless one is nonexistent). Although immaterial to the compiler, it is recommended that this convention be rigidly adhered to in all nested comparisons and in all but the simplest single comparisons. Examples are

```
A = B:
    C ≠ D:
        1 → E;
        2 → E;;
    3 → E;

A = B:
    C = D:
        E ≤ 4:
            5 → F;
            6 → F;;
        7 → F;
    A - 4 → I, SUBROUTINE;
    A > B:
        A → B;
        -B → B;;
```

NELIAC-N permits the use of up to 15 <u>active</u> nested comparisons at any one time.

BOOLEAN OPERATORS

The Boolean operators of AND ∩ and OR ∪ may be used to string a number of these comparisons in a statement, as long as only one type of operator is used in such a statement. Note the following examples:

```
DIMENSION FLAG = 0:
   NEXT OPERATOR ≠ COLON ∩
   LEFT BRACKET < CURRENT OPERATOR < RIGHT ARROW:
      SET OPERAND.
      TEST FOR PASS COMMAND;;;

A < B ∪ C < D ∪ F ≠ K:  TRUE; FALSE;
```

Note that a statement of the form:

```
A < B ≤ C ≠ D: TRUE; FALSE;
```

is really a series of and statements; namely:

```
A < B ∩ B ≤ C ∩ C ≠ D:  TRUE; FALSE;
```

Hence, compound statements of this type may only be linked with a series of Boolean and comparisons and not with a series of Boolean or comparisons.

In a group of nested comparisons though, the form of each individual comparison statement is independent of the forms of all the other comparison statements.

A string of and comparisons may contain up to 16 individual comparisons; a string of or comparisons may contain up to 15 individual comparisons. Since there are different restrictions on the permissible forms of the left and right sides of a comparison statement, they must be defined for Boolean strings. The exact definition is that a right side begins immediately after one of the six relational operators and is terminated by the next colon, Boolean and, or Boolean or. In the case of a Boolean and or Boolean or, a new left side then begins. In the case of a statement like $A < B \leqslant C < D$: the right side restrictions apply to all quantities except A.

VI. SUBSCRIPTED VARIABLES

Suppose, as an example, we wish to compute the sum of the squares of fifty numbers, X_0 to X_{49}, and store the result in SUMSQ. Each element in this table of fifty variables may be called upon by subscripting the name of the table X. Subscripting is accomplished by the use of brackets [] surrounding the number indexing the individual element of the table. Remember, in NELIAC, subscripting begins at <u>zero</u> and not one; thus X [0] would refer to the first value of the table while X [49] would refer to the last; i.e., the fiftieth.

Indexing also may be done via one of the 6 index registers of the compiler, referenced by the names I, J, K, L, M, and N or by any fixed point variable dimensioned by the programmer. These registers may be treated in a manner similar to any <u>fixed point</u> variable. Within the program logic, therefore, an element in a table may be referenced by X [I] and the index register I augmented as necessary.

The most general form of subscripting in NELIAC-N is

OPERAND [SUBSCRIPT ± number]

The exact address or location represented by this expression is obtained as follows: take the address of the name OPERAND as the base address, add to it the address currently contained in the location identified by the name SUBSCRIPT, and add or subtract (as the case may be) the explicit value of number. The resulting address is the address of the variable being referenced by the given expression. In the expression, OPERAND may be any name dimensioned in the program, SUBSCRIPT may be any fixed-point entire-word noun dimensioned in the program (including the index registers I, J, K, L, M, and N automatically dimensioned for the programmer), and number may be any unsigned fixed point integer - decimal or hexadecimal. In this general expression, all degenerate cases formed from the suppression of any one or any two of the three quantities involved are valid forms having the meanings immediately derivable from the general form. The case where the variable OPERAND is suppressed is covered in the chapter on ADDRESSES OF NAMES.

With this information, we may illustrate one method of accomplishing the sum square problem.

```
BEGIN:
0 → I → SUMSQ,
COMPUTE SUMSQ:
X [I] * X [I] + SUMSQ → SUMSQ,
I + 1 → I = 50:  EXIT.  COMPUTE SUMSQ.

EXIT: ..
```

All subscripting is accomplished by variables, including the index registers, and/or fixed point constants, though, of course, the values in the table being subscripted may be all fixed point or all floating point.

Legal subscripted variables:

```
MAST [2]
X [J]
TNT [K + 2]
Z [J - 3]
W [INDEX]
Y [NAME - #300]
V [-50]
```

In general, subscripted variables are treated just like ordinary variables. For example, they may be used in arithmetic expressions:

```
A [I + 2] + B [J - 3] / C [10] → D [M]
```

and on either side of a comparison statement:

A [I] + B [L+3] ≤ C [10] : TRUE; FALSE;

etc...

SUBSCRIPTED STRAIGHT JUMPS

One useful feature of the NELIAC language is that of the Jump Table, another method of branching within the program logic. Jump tables are defined, within the program logic, by punctuation, a unique name (which is thus a verb), a colon, and a series of jump commands.

, JTABLE: JUMPA. JUMPB. JUMPC.

A jump command to an element of this jump table may be written as

,JTABLE [I].

which indicates an unconditional jump to the Ith element of the jump table which is, in turn, a command to jump to a portion of the program defined elsewhere. For example, if the value of index I = 0, the above command will cause a jump to JUMPA, etc....

Subscripting may be applied only to straight jumps; i.e., jumps to entry points, and may not be applied to return jumps; i.e., subroutine calls and function calls.

SUBSCRIPT PACKAGE

In NELIAC-N, subscripting by name is accomplished through a return jump to the subroutine SUBSCRIP contained in the subscript package which is automatically compiled into any program requiring it. Hence, this name must not be used by the programmer.

VII. LOOP CONTROL

Perhaps one of the most useful features of today's high speed computers is the capability of repeating certain operations; i.e., the procedure remains the same, but the variables used are different. This objective may be accomplished in NELIAC by the use of LOOP CONTROL, a method of indicating the procedure to be followed and the specific number of times it is to be executed. The use of loop control along with that of subscripted variables provides a powerful tool in computation. Consider the following example.

```
, J = 0 (1) 24 { P [J] + Q [J] → TAB [J] }
```

The procedure to be repeated is enclosed within braces, with the loop control preceding. Conventionally, one of the index registers (I, J, K, L, M, and N) is used for loop control and subscripting although any other full-word integer variable may be used just as efficiently. The statement above reads that the index register J is set to zero and the procedure executed for the first time; thus, the first value of the table P; i.e., P [0], is added to the first value of the table Q; i.e., Q [0], and the sum is stored into the first cell of the table, TAB [0]. The index register J is incremented by 1 and the loop repeated this time using variables P[1], Q[1], and TAB [1], etc...., until 25 values (corresponding to the subscripts 0 to 24) are added and stored into the 25 locations of table TAB. Optionally, the parenthetic ALGOL word FOR may be used for clarity in the printed copy. In that event, the above example becomes

, FOR J = 0 (1) 24 {P [J] + Q [J] → TAB [J] }

Let us look closer at the basic format of the loop control.

FOR	ALPHA =	BETA	(GAMMA)	DELTA	{PROCEDURE}
ALGOL Word	The Controlling Word or Loop Parameter	Lower Limit of Loop	Incrementing or Decrementing Steps	Upper Limit of Loop	

1. The ALGOL word FOR in the loop control is optional and is used only for added readability. It is actually ignored by the compiler.

2. ALPHA is the controlling word of the loop control. It is conventionally an index register though a fixed point full word variable may be used just as efficiently. Note that the value of ALPHA may be used as a subscript within the procedure.

3. BETA contains, or indicates, the first value of the controlling word. It may be a fixed point integer, a fixed point variable name, another index register, or any one of these ± another, ad infinitum; i.e., BETA consists of a theoretically unlimited string of sums and differences of unsigned, unsubscripted, and unbithandled fixed point variables and unsigned integer constants.

4. GAMMA, the incrementing or decrementing steps to be taken, may be a fixed point integer or a fixed point unsubscripted, unbithandled variable containing a positive integer; the latter may be accompanied by a negative sign (see Note below).

Note: The full meaning of item 4 above should be clarified. It is legal to decrement in the following manner.

```
FOR I = A(-1) 0
```

using the explicit value of -1. However, it is illegal for

GAMMA to be a variable that contains an integer equal to or less than zero. Hence, if the value in DEC is -1, then:

```
FOR I = A(DEC)0
```

is illegal. On the other hand, if DEC were to contain +1, then the following is legal:

```
FOR I = A(-DEC) 0.
```

5. DELTA, the last limit of the loop, may take any of the forms of BETA.

6. The procedure itself may be any legal set of statements ordinarily used within the program logic, including return jumps to subroutines, comparisons, additional loops with different loop parameters, etc.

From these rules, we can see that all of the following formats of loop control are legal.

```
,J = A+B (-1) 0 {        }
,K = I (5) COUNT {       }
,M = NUMBER + 10 (-2) K + 1 {       }
,NOUN = 5 (NN) FINISH -1 {      }
, I = I (1) END {       }
```

The number of loops executed will never continue beyond the limit of DELTA. A simple example will serve to illustrate this point.

```
, FOR I = 0 (2) 5 {       }
```

Obviously, the count will never hit 5; one might expect the loop to continue indefinitely. However, this is not the case. The loop will be executed, and whenever incrementation by 2 will cause the count to be greater than 5, the loop control will be terminated. Thus, the preceding loop will be executed three times; i.e., for I = 0, 2, and 4. After the completion of any loop, a normal exit will occur and the next sequence of instructions will be executed. Similarly, if the loop control is being decremented, the program will never be operated for a count less than DELTA.

In NELIAC, considering the general loop control statement given in this chapter, the loop increment GAMMA and the upper limit DELTA are variable; i.e., if either or both are altered by the procedure within the loop braces, the new value(s) of the loop increment and/or upper limit will be used until altered again. The same condition exists with respect to the loop parameter ALPHA; i.e., it is this altered value of ALPHA which will be used throughout the remainder of this repetition of the loop and which, furthermore, will be incremented or decremented at the end of the repetition. Finally, although alteration of the lower limit BETA by the procedure within the loop braces will not affect the further repetitions of the procedure during this execution of the loop control statement, if, at a later time, control is again transferred to the loop control statement, the new value of BETA will be the value then considered as the lower limit of the loop parameter (assuming BETA has not been changed again elsewhere in the program).

The value of the loop parameter ALPHA upon exiting from the loop is its value during the last execution of the procedure within the loop braces (assuming the procedure does not alter it).

Let us rewrite the program logic of the previous example to compute the sum of the squares of fifty values of X_0 to X_{49}, assuming that the number of variables in table X has been defined in the dimensioning statement NR VALUES = 50, as:

Thus, that portion of the program to compute sum squares might read:

```
COMPUTE SUM SQUARES:
0 → SUMSQ, FOR K = 0 (1) NR VALUES - 1
{ X [K] * X [K] + SUMSQ → SUMSQ } ..
```

VIII. FUNCTIONS

In loop control, the method of indexing tables of values for computation in similar operations was illustrated. Other instances, however, may call for an operation to be performed several times with different parameters but at individual points in the program; e.g., a common routine to compute square roots may be necessary. In cases such as this, the NELIAC function notation may be used. This functional notation enables the programmer to execute a particular procedure with any desired input parameters necessary to determine the value(s) of the function with the result(s) being placed into any desired output parameter(s). Though the function is defined but once, it may be executed at any point of the program logic (except within itself, of course). With the exception of its parameters, a function is written and executed in a manner similar to a subroutine.

An example of the format of the functional <u>definition</u> is:

```
PROCEDURE X (W. Y. Z.);
{W * W → Y * W → Z }
```

The function name is any unique name followed by its associated dummy parameters enclosed within parentheses. As with a subroutine, a colon precedes the computational logic which must be enclosed within braces. This computational logic may contain all computational procedures which are valid in the main program except (1) subroutine and function definitions and (2) calls for itself though calls for any other subroutine or function are valid.

A function, written in proper notation, must indicate the mode of both input and output parameters although the distinction between input and output parameters need not be indicated here. In fact, in the function definition this distinction can be indicated to the reader only, not the compiler, since the distinction is actually made only in function calls. The arguments within the parentheses serve the same purpose as the dimensioning statement of a program (or flowchart); thus, anything legal within a dimensioning statement (except absolute addressing, see the chapter ADDRESSES OF NAMES) is legal within the parentheses. As usual, a comma after fixed point variables suffices, and here too it is also legal to define floating point variables with a period only. The variables (within the parentheses) in a function definition are merely dummy names and, therefore, names local to the function subprogram; thus, the same names may be used elsewhere in the program without harm, although this is usually inadvisable since it complicates debugging, understanding, and altering the program. The instructions within the braces are equivalent to the program logic. In fact, the function may be considered as a miniature flowchart accessible only through its name.

Again, as with a subroutine definition, the function definition does not cause computation to take place. Execution occurs when the function is called within the program logic by writing the function name and specifying the actual arguments (parameters) to be used. It is here, and here alone, that the compiler is told which parameters are to be treated as input and which as output. Note the following example which executes (i.e., calls) the function, PROCEDURE X, previously illustrated.

```
, PROCEDURE X (ARG; ANSWER, ANSWER [1] ),
```

The parameters supplied must agree exactly in mode, order, and number as anticipated by the function definition. Commas separate the parameters since indication as to mode is unnecessary (in fact, meaningless) in the calling of a function; the manner in which these variables are treated is completely determined in the function definition. A semicolon separates the input arguments from the variables specified for the output of the function. In this case, the comma normally used after a parameter must be replaced by the semicolon since its usage here in addition to the semicolon would not be redundant but would have special meaning as will be seen later.

The arguments thus supplied as input parameters are substituted for the corresponding dummy variables in the definition, the values of the function are computed, and the values of the dummy variables in the definition are inserted into the corresponding arguments supplied as output parameters. As a result of the above call for PROCEDURE X, ANSWER will be expected to contain the value of ARG squared, ANSWER [1] the value of ARG cubed.

As an illustration of legal parameters which may be used in a function call, note the following example:

```
FUNCTION Y (A, B[I], C[4]; D[K+2], E[F-#300] (16→19)),
```

The bit notation used in the last parameter will be described in a later chapter. An example of the definition of dummy variables which may be used when writing a function follows:

```
XFNCT (X = 0*0, Y(25), D. A = {B}. C: D: {E(24→31), F(24→47)},
        G = 17.578):
            {Program Logic}
```

The unfamiliar forms of dimensioning will be described in later chapters.

As has been stated, functions are merely sub-programs in which the variables within the parentheses are equivalent to the dimensioning statement and the program logic is contained within the braces. There is no limit to the number of input parameters which may be entered in a function definition nor is there a limit to the number of output values which may be computed. However, every function must have at least one input parameter though it need have no output parameters. Functions, just as subroutines, should be defined at the end of a program or its flowcharts, or necessary jumps should be made over the function segments of the program. In the following section, we shall learn a method whereby functions and subroutines may be written as separate flowcharts, virtually independent of the main program.

In a function call, the most general forms of the input parameter are (1) the unsigned general subscripted, bit-handled noun and (2) any unsigned legal form of a constant in program logic. The most general form of the output parameter is form (1) of the input parameter.

The one basic concept which must be grasped in functional notation is that the correspondence between the arguments used as parameters in a function call and the formal parameters dimensioned in the function definition is solely on the basis of their respective ordering starting with the first parameter in each case. If a parameter is defined in a function definition and it is desired not to utilize this parameter in a particular function call, this fact must be indicated to the compiler by leaving a blank space between the commas (one of which may be a semicolon instead of a comma) where the argument corresponding to this formal parameter would normally be placed (unless no further parameters in the ordering are to be utilized). Suppose a function is defined as follows

, FUNCTION (U, V. W. X. Y. Z): { Program Logic }, Then, the function call

```
, FUNCTION (7, 6.341, A [J-4]; B, C[D], E[2]),
```

will result in the input parameters 7, 6.341, and A[J-4] being placed into the formal parameters U, V, and W, respectively, before execution of the procedure defined as FUNCTION, and the formal parameters X, Y, and Z being placed into B, C[D], and E[2], respectively, after execution of the function. However, if it is desired to call the function leaving the formal parameters U and W unchanged and only securing, as output, the value of the formal parameter Y, the function call may be written as

```
, FUNCTION (, 1.0*6, ; , F),
```

Comparing this function call to the function definition, the reader will easily see, solely on a basis of ordering, that the parameter U will be unchanged, a floating point one million (1.0*6) will be placed in parameter V, parameter W will be unchanged, the procedure defined as FUNCTION will be executed for these values of U, V, and W, then the value calculated and placed in X will be ignored, the value calculated and placed in Y will be placed in F for use in the main program, and the values calculated and placed in the remaining parameters; namely, Z, will be ignored.

IX. PROGRAM STRUCTURE

So far, NELIAC programs have been described in terms of a single load number, dimensioning statement, semicolon, program logic, and double period. Actually, complex programs often consist of several such sub-programs, called flowcharts. Each separate flowchart must follow this format headed by leader and followed by leader:

```
(Leader)
5
DIMENSIONING STATEMENT
;
PROGRAM LOGIC
..
(Leader)
```

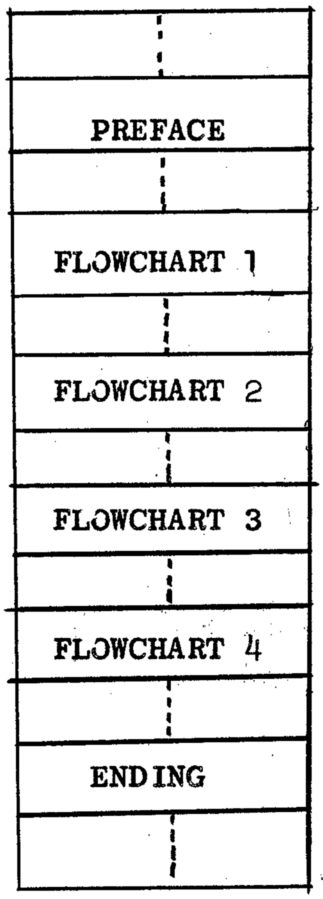
One or several flowcharts (with a maximum of 63) preceded by a preface and followed by an ending comprise a program. The preface consists of:

```
(Leader)
5
(Optional comments)
Program or Programmer's Name,
Object Program First Address, Bias ..
(Leader)
```

Either or both the Object Program First Address and the Bias may be left blank in which case standard addresses will be used for the blanks. The ending consists of:

```
(Leader)
5..
(Leader)
```

A NELIAC program tape consisting of 4 flowcharts may be represented schematically as (without any attempt at relative scaling):



Obviously, the ability to write programs as separate flowcharts allows one to eliminate the necessity of having to bypass subroutines and functions within the main program logic. However, an even more important reason for this structure is to permit the name purge feature which is described in the next paragraph. As shall be seen, this feature provides a solution to many of the problems encountered when several programmers are engaged in writing different parts of the same lengthy program.

Suppose, e.g., a programmer wishes to use a subroutine which already has been written by someone else at some other time. Obviously, a problem may arise in duplication of names, as the programmer must avoid using any names already defined in the subroutine. In NELIAC, this problem is greatly diminished, since the writer of the subroutine can purge names that have no significance outside the flowchart containing the subroutine. Names thus purged may be used for other purposes in the remaining flowcharts. For example, a square root subroutine would have virtually all names purged. The only names not purged would be the ones necessary to communicate with the main program in a separate flowchart. In fact, the use of functional notation, rather than subroutine notation, completely eliminates the need for even these names.

Purging is accomplished by inserting an absolute sign | anywhere within the name as it is being defined (but not inserted when the name is used) although, conventionally, it is placed after the first character of the name.

Purged names within the Dimensioning Statement:

```
I|NDEX = 6,
T|1,
X| = 0 * 0,
```

Purged names within the program logic:

```
, C|ONT : A → B,
, C|LEAR : {0 → I → J → K}
```

To reiterate, these names, known as <u>temporary</u> or <u>local</u> names, will have meaning only in the flowchart where the above definitions occur.

Now that it is possible for a program to consist of more than one flowchart, it also becomes possible for a dimensioning statement to follow part of the program logic of the program. This possibility necessitates the following programming rule:

> Each floating-point, partial-word, and IO format and IO subscript variable must be defined in a dimensioning statement (or function definition) before it is used in any program logic.

Partial words and IO are discussed in later chapters. This rule is necessary because the NELIAC compiler must distinguish between the two number formats, floating point and fixed point, when making up instructions pertaining to a variable in the program logic. Corresponding necessities arises in the case of dimensioned partial words and in the case of format and subscript words referred to in IO statements.

For example, suppose a programmer wishes to write his main program as the first flowchart, and include a random number generator subroutine (called RAND) as the second flowchart. The pattern is illustrated below:

```
        (Leader)
        5
D.S. 1  DIMENSIONING STATEMENT FOR MAIN PROGRAM
        ;
        MAIN PROGRAM LOGIC
        ..
        (Leader)
        5
D.S. 2  DIMENSIONING STATEMENT FOR THE RANDOM
          NUMBER GENERATOR SUBROUTINE
        ;
        RAND: {PROGRAM LOGIC FOR RNG SUBROUTINE}
        ..
        (Leader)
```

Suppose the random number generator stores its random number in floating point in location X just before exiting. Since the main program is going to use X, X itself must be defined as a floating point variable in D.S. 1. It would be illegal to define X as floating point in D.S. 2 because in that case the main program would be compiled before the compiler was able to <u>sense</u> that X was to be floating point. Of course, the way to get around this problem is to write RAND as a function, defining the output with a dummy output floating point name as follows:

```
(Leader)
5
DIMENSIONING STATEMENT FOR RNG SUBROUTINE
;
RAND (Y; DUMY.):{(generate a random number) → DUMY }
..
(Leader)
```

Then with the above RAND function as the second flow-chart the following call in the main program logic will generate a random number in location X (where X must be defined as floating point in D.S. 1.):

RAND (;X),

The dummy input parameter Y is used simply because every function must have at least one input parameter.

Appendix D is the current version of the NELIAC-N Coding Sheet used by the programmer for writing NELIAC programs for the NAREC.

Appendix E is the current version of the NELIAC-N Operator Instruction Sheet filled out by the programmer and transmitted to the NAREC operating staff for compilation (and possible run) of his NELIAC program on the NAREC.

COMPUTER SPACE LIMITATIONS

Although the NELIAC language itself places no limitations on such features as number and size of flowcharts, number of names, number of undefined calls, length of object program, etc., the version of the language implemented for a particular computer must, of course, be limited by the space limitations of that computer's memory. Most of the limitations such as names being uniquely defined in their first 16 character, the limitations on nested comparisons and strings of Boolean or and Boolean and statements, etc., which have already been described are due to hardware limitations rather than NELIAC language limitations. In addition, the NAREC imposes limitations on the overall characteristics of NELIAC-N just as every computer does to the version of NELIAC implemented on it.

NELIAC-N allows the compilation of up to 63 flowcharts in a single sweep. However, there is an IO Package and a Library Package which are compiled individually as separate flowcharts at the end of the programs requiring them. Since either or both of these flowcharts may be added to a program, the programmer's flowcharts may actually be limited to 61 or 62. These two package flowcharts will be discussed in greater detail in the chapters devoted to them. The fixed point, floating point, and subscript packages are each compiled individually at the end of the first flowchart requiring the particular package but as parts of those flowcharts. Thus, they impose no such limitation on the source program.

Immediately upon readin, the NELIAC-N flowchart is converted to a symbol string containing, in order, the NELIAC characters of the flowchart converted to an internal code in which there is a one-to-one correspondence between the NELIAC characters of the flowchart and the symbols of the symbol string. In this symbol string, all spaces external to names and numbers have been removed, successive spaces within names and numbers have been reduced to single spaces, and all ALGOL words have been eliminated, but comments have been retained. The storage area allocated to this symbol string limits the length of the flowchart when reduced to its symbol string to 5600 characters at the present time. This normally allows from 5 - 15 flowchart pages depending upon the character density of the pages.

In the event that this limitation is exceeded, the computer will stop with a Flowchart Area Overflow fault printout. However from many other standpoints - understanding, debugging, correcting, changing, combining, etc., of flowcharts, it is advisable to write flowcharts of individual length far below this overall limitation.

The NELIAC-N compiler contains a list of 512 entries in which all names, constants and masks used in logic and IO statement entries are recorded. Temporary names are recorded in the list but are purged from the list at the end of their flowchart thus making their space available for reuse. Since, to date, no program including the compiler itself, has ever overflowed this list, it is considered more than adequate for any foreseeable program. If the list is overflowed, a Name List Overflow fault printout will result.

The NELIAC-N compiler contains a list of 300 locations for recording the names, constants, and masks as yet undefined. Since each location can record two entries for the same name, number, or mask, 300-600 undefined calls are permitted at any one time. Whenever a name, number, or mask is defined, all undefined calls for it are filled in the object program and purged from this list thus making available this space for reuse. Since constants and masks are defined at the end of the flowcharts where they are first used, they will be undefined throughout the first flowchart where used but defined throughout the remainder of the program. Since subscripting by name, fixed point multiplication and division, and floating point addition, subtraction, multiplication, and division are performed through return jumps to subroutines in packages compiled at the end of the flowcharts where first required, these operations will set up undefined calls in the first flowcharts where these operations are used. Hence, this procedure provides another reason for writing NELIAC programs in relatively short flowcharts. In the event that this list is overflowed, an Undefined Name Overflow fault printout will occur.

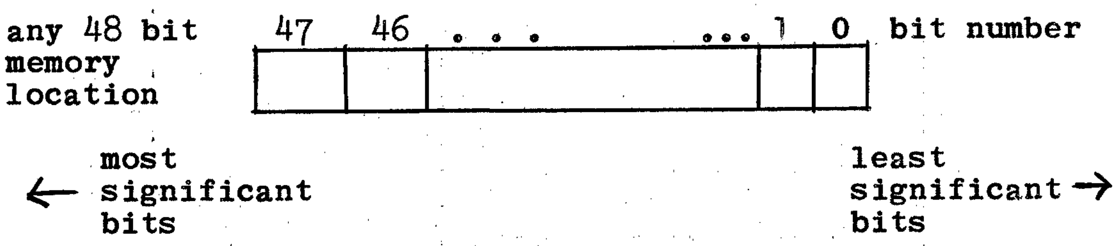
Finally, since the compiler itself, at the present time, occupies memory locations #0000 to #26FF in the NAREC, this leaves the area #2700 to #3FFF available for storage of the resulting object program as the NELIAC program is being compiled. Hence, normal compilation allows for object programs up to #1900 or 6400 locations. However, "reset the bias" and "low standard bias" features allow the compilation of larger programs (such as the compiler itself which occupies 9984 locations) in a single sweep. In addition, by suitable use of absolute addressing, a program may be compiled in two or more sweeps. If the resulting object program ever exceeds the area available for its storage, the NAREC will stop with a #4000 42 in the control register.

X. PARTIAL LOCATION OPERANDS (BIT HANDLING)

Up to this section all storage variables have been discussed in terms of a full 48 bit word or memory location per variable. In this section we shall see that any continuous portion of a memory location (i.e., only selected bits) can be defined as a fixed point integer variable, and that in the program logic, any continuous portion of a variable can be manipulated quite easily without disturbing the rest of the bits* of the memory location to which the variable is defined.

*NOTE: Conventionally, the term bit is the name given to each of the 48 flip-flops which, together comprise a NAREC memory location. This name is derived from Binary Digit because it can contain either of the values 0 or 1.

It will be convenient to use the following <u>bit number</u> assignments:



<u>PART VARIABLE OPERANDS</u>

The reader is already familiar with the procedure for defining a full 48 bit fixed point integer variable (Chapter II). If the programmer wishes to manipulate only selected bits of such a variable he specifies the name of the variable, and indicates which group of bits of that variable he wishes to treat as a <u>positive</u> fixed point integer* by writing the first (lowest bit number) and last (highest) bit number using parentheses and the right arrow as illustrated:

$$A\ (0 \rightarrow 14)$$

*The integer is necessarily positive only when referring to 44 bits or less.

In this example only bits 0 through 14 of the variable A are referenced. To call for a single bit, say the least significant, or bit zero, one would write:

$$A\ (0 \rightarrow 0)$$

If the variable is part of a table of variables and requires a subscript for its reference, the subscript notation (the brackets) is written first; i.e.,

$$A\ [I]\ (6 \rightarrow 32)$$

It should be noted that the values of part variable operands of less than 45 bits are treated as positive fixed point integers whereas full 48 bit variables may contain either positive or negative integers. In the case of part words of 45 to 48 bits, whether the part word is considered positive or negative depends on the setting of the sign bit - bit 44 in the NAREC - after the part word is downshifted so it begins at bit 0. For example, supposed variable A contains the following array of bits:

		47 … 7	6	5	4	3	2	1	0
A	0	0 0	1	1	0	1	0	1	1

One immediately recognizes this as the integer +6b (hexadecimal) or +107 (decimal). However, the 4 bit operand A(2→5) which contains the binary array 1010 is considered to contain the number a (hexadecimal) or 10 (decimal).

In other words, if one were to write the following program (assuming A is defined as above):

```
,A (2→5) = #a: I + 1 → I;; STOP.
```

the result would be that the program would add 1 to I. Of course, an equivalent statement would be:

```
,A (2→5) = 10 : I + 1 → I;; STOP.
```

from which the compiler would generate the same program. It is worth reiterating that even though the uppermost bit of A(2→5) (bit 5 of variable A) is a 1 the partial operand is not considered to be a negative integer. The only possibility of the partial operand being considered a negative integer in NELIAC-N is if it contains more than 44 bits.

All arithmetic operations previously described for fixed point operands are legal with part variable operands. However, the responsibility of arranging adequate storage capability is left to the programmer. For example:

Legally, the programmer may write:

,TABLE[I](19→25) * A(3→6) → Z(1→5),

However, the programmer should realize that a 7 bit operand times a 4 bit operand may require as many as 11 bits to store the answer. In the above case, only the lower 5 bits of the answer would be stored into Z(1→5), and the upper 6 bits would be lost.

The index registers, I, J, K, L, M and N, automatically dimensioned by the compiler may be bit-handled exactly the same as any noun dimensioned in a dimensioning statement. Thus:

,I → T (13 → 18),
,Z (41 → 46) → K,
,M (5 → 10)/2 → L (24 → 31),

Further operations with part variable operands are illustrated below:

```
,A(25→34) → B,
,A(39→47) + B(0→14) → C,
,A(12→24) < B(2→14) : TRUE ; FALSE ;
,A[I](30→36) → B[J](31→35) → C[K](0→14),
,A(44→44) = 0:  TRUE ; FALSE ;
```

PART LOCATION VARIABLES

The above discussion shows how any portion of a variable may be manipulated without disturbing the rest of the bits of the variable. It is possible, and often much more convenient, to define a variable as certain bits of another variable. Since they reference only part of a 48 bit memory location, they are called PART LOCATION VARIABLES, and are considered to be variables, themselves. Part location variables are always defined as certain bits of a variable which itself is defined as an ordinary full location variable (although this variable need not be explicitly named and dimensioned). For example, if X is to be bits 39 to 47 of variable A, one would define this in the dimensioning statement, along with the definition of A, as follows:

```
A: { X(39 → 47) },
```

In the program logic which follows, the operands A(39→47) and X would be indistinguishable, and all the rules for part variable operands described in the previous section would apply to the part location variable X. Obviously, the main advantage in using part location variables is to pack a number of variables whose ranges of values are small into the same memory location. An illustration of a typical use of a table of packed part location variables follows:

Suppose we wish to store data on 100 aircraft. Items we wish to store are:

X coordinate (15 bits)
Y coordinate (15 bits)
height in 1000 ft. units (6 bits)
status (3 bits)
identity (3 bits)
track number (6 bits)

This data can be packed into 100 48-bit words of NAREC memory as follows: In the dimensioning statement one would write:

```
AIRCRAFT: {X(0→14), Y(15→29),
HT(30→35), STAT(36→38), ID(39→41),
TN(42→47) } (100),
```

Note that the initial value of all of these variables is zero. So far, there is no convenient way to set all part location variables to desired initial values since only entire words may be assigned non-zero initial values. Hence, in order to dimension initial values for this table, X, Y, HT, STAT, ID, and TN would have to be combined into the full word AIRCRAFT for each entry in the table. Then, of course, the initial values will be assigned in the normal manner for tables. An alternate solution would be to use constants in the first part of the program logic; i.e., 3052→X[0], 20 425→Y[0], etc...

Note also that each of the part location entries in the table AIRCRAFT: X, Y, HT, STAT, ID, and TN, are tables of 100 variables. Thus to reference the X coordinate of the 10th aircraft one would write X[9] (or equivalently AIRCRAFT [9](0→14)).

Before leaving this example, it is well to illustrate a technique that often makes the program logic easier to read. Suppose the programmer wishes to distinguish between 4 identities, FRIENDLY, HOSTILE, FAKER, UNKNOWN. The programmer might arbitrarily assign values 0, 1, 2, and 3, for these 4 identities respectively, and then in the program logic, if the program wishes to find out if a certain track has identity of FRIENDLY, the program might read:

```
, ID[I] = 0:  YES.  NO.
```

However, a preferred method is to define variables FRIENDLY = 0, HOSTILE = 1, etc., in the dimensioning statement and then the same program could read:

```
, ID[I] = FRIENDLY :  YES.  NO.
```

Of course, not all bits of a full 48 bit variable need be dimensioned, and several names may be given to the same bits of a full variable. Part location variables of the same full variable may overlap each other:

```
B: { C(0→12), D(0→12), E(12→29),
     F(12→47) },
```

Furthermore, the entire word need not be named and defined:

```
{C(0→12), D(0→12), E(12→29), F(12→47)},
```

XI. OUTPUT STATEMENTS

The NELIAC-N compiler converts NELIAC output statements into print programs that are compatible with the on-line printer system or with the off-line NELIAC-N Flexowriter (through the output punch).

In general, each NELIAC output statement controls the printing of a single line of print of up to 72 characters for the line printer or 86, 116, or 160 for the flexowriters. Output statements are also used to specify line spacing, paging, and termination of output.

Two types of printed output control are required by the programmer: first, he must have the ability to specify the format of the data he desires to have printed, and second, he must have a method of printing literals; i.e., any words or symbols verbatim to serve as headings, labels, or lines of text.

The information a programmer must supply pertaining to his printed data consists, first, of specifications about the data itself:

1. Which variables are involved and in what order are they to be printed?
2. Are the numbers to be printed fixed point or floating point variables, and, if fixed point, should they be printed in hexadecimal or decimal notation?
3. How many digits to the right of the decimal point are required for floating point variables?

Secondly, indication as to the arrangement of such data upon the printed page must be made:

1. How many spaces are needed between each piece of data on a single line?
2. Are blank lines needed?
3. Are new pages needed?
4. When is the output terminated?

PRINT VARIABLES

The term *print variable* will be used here to mean a variable whose value is to be printed through the use of an output statement. Only full 48 bit variables can be used as print variables. The basic format of an output statement as it is written within the program logic will now be examined. In this section, only the control of print variables; i.e., data printout, will be considered.

The essential elements of a print statement are a *comma* and a *left brace*, the names of the print variables enclosed by the *less than, greater than* signs, and the *right brace* indicating the completion of the statement. Such an output statement will print *one line* only. Consider the example below in which the two variables, referenced by name as DATA1 and DATA2, are printed on a single line.

, {PRINT < DATA1 | DATA2 > } ,

The word PRINT is merely a mnemonic device which may be omitted. In fact, any words may be inserted here without harm although it is not customary to insert anything. Spaces between data words are indicated by the <u>absolute</u> sign |, the Boolean or sign ∪, and the Boolean and sign ∩. The absolute sign indicates one space; the OR sign indicates five spaces; the AND sign indicates no spaces. Thus, three spaces are indicated by ||| and eleven spaces may be indicated by a combination of the two symbols ∪∪|, |∪∪, or ∪|∪. A Boolean <u>and</u> sign ∩ is necessary if no spacing is required between print variables.

We see that the output statement serves only to indicate the print variables, the spacing between printed values, and, by its position in the program logic, when the line is to be printed. All other control over the printed message is indicated by the programmer in the dimensioning statement. Thus, for each <u>print variable</u>, the programmer must indicate in the dimensioning statement the desired printed number format (scientific or fixed point), the number system to be used (hexadecimal or decimal), and the number of digits to be printed, (which also controls the total number of print spaces used every time the variable is printed).

The number of digits to be printed is the same as the number of digits in the initial value (with the exception of certain conventions); i.e.,

A = 50,

would specify two printed digits. The number of spaces required would be three, however, as a space is always reserved for the sign of all print variables except for a full 12 digit hexadecimal word.

B = #00,

specifies a printing of the sign, the hexi sign, and the least two significant hexadecimal digits (after complementation if the word is negative) thus requiring four print spaces. The sign of a value is actually printed only if the value is negative.

Floating point print variables require an additional space for a decimal point, and, in scientific (true floating point) format, five additional spaces for an exponent.

Floating point print variables can be printed in either scientific or true decimal point format. Scientific format is always printed with a fraction part, X, where $1/10 \leq X < 1$, and a signed power of 10 expressed as a plus or minus integer in three digits. To indicate scientific format in the dimensioning statement, an initial value is written without a decimal point. For example, if A is defined as:

A = 0000 * 0,

then if the floating point number 23.14 were stored in A and printed, the resulting output would read as:

.2314 -002

and thus would use a total of 11 spaces on the printed output page.

True decimal point format for floating point variables is always printed with an appropriately placed decimal point. Thus if B is defined as:

B = 0000. * 0,

then if B contains a floating point value of 269.733, the printed result would read as:

270.

In all cases the decimal point is printed.

All values printed from a table of variables will be printed with the same format control. This control will be determined by the <u>last</u> specified initial value of the table; e.g., a table may be defined in the dimensioning statement as:

A(3) = 295, 23, 48,

Since the last value in this table is 48, only two digits have been specified for any print variable in the entire table. Any output statement calling for the printing of variable A (the first value of the table in this example, 295) will print <u>only</u> asterisks since the value of A is too large for the dimensioned format of A. Hence, if the program logic were to read:

, { < A > } ,

the printed result would be:

* * *

It is good practice to format a variable larger than the greatest expected value to allow for any miscalculation. Neither fixed point print variables nor floating point print variables, when larger than the specified format, will be printed. A row of asterisks is printed instead of the number.

As another example, if a table is already defined in the dimensioning statement as:

```
P MATRIX (9) = 13.21 * 0, 2.32 * 0, 1.00 * 0,
                        ,-0.98 * 0, 0.75 * 0,
                        ,          ,00.34 * 0,
```

and if it is desired to print this table as it stands, two zeroes should precede the decimal point of the last value (00.34 * 0) to enable the printout of the first values (13.21 * 0). The table may then be printed in the following manner:

```
FOR J = 0(3)6 {, { < PMATRIX [J] |
        PMATRIX [J+1] | PMATRIX [J+2] >} }
```

As an example, let us suppose a table has been formatted in the following manner:

```
TABLE (4) = 0000.00 *0,
```

and floating point variables are computed and stored into this formatted table. Output statements may be enclosed in loop control statements so that an instruction in the program logic may read:

```
, FOR I = 0(1)3 {,> < TABLE [I] > } }
```

The table, printed out, may appear as:

```
    2.01
  -14.32
   -3.75
********
```

The value of the last variable was too large for the allotted format; i.e., over the value 9999.99 after round-off, therefore, the asterisks.

More than one line of print may be specified. The following example illustrates an output statement indicating three lines of print, two variables per line.

```
, { < A|B > < C|D > < E|F > },
```

On the next page, the foregoing discussion is illustrated by indicating sample dimensioning statements, the number contained in each print variable when the output statements were operated, and the resulting NELIAC printouts.

FIXED POINT

Dimensioning Statement:
A = 0, B = 00, C = 00000, E = #0, F = #00000, G = #,

Number	Neliac Printout					
492	**	***	492	***	#001ec	#00000001ec
32765	**	***	32765	***	#07ffd	#000000007ffd
-30	**	-30	-30	***	-#0001e	#ffffffffe2

FLOATING POINT

TRUE DECIMAL POINT

Dimensioning Statement:
AA = 0. * 0, BB = 0.0 * 0, CC = 000.00 * 0,

Number	Neliac Printout		
1.	1.	1.0	1.00
-1.	-1.	-1.0	-1.00
371.21	***	****	371.21
-371.21	***	****	-371.21

SCIENTIFIC NOTATION

Dimensioning Statement:
GG = 25 * 2, HH = 2314 * -2, II = 0 * 0, JJ = 00000 * 0,

Number	Neliac Printout			
	.25 +004	.2314 +002		
1.	.10 +001	.1000 +001	.1000000000 +001	.10000 +001
-1.	-.10 +001	-.1000 +001	-.1000000000 +001	-.10000 +001
371.21	.37 +003	.3712 +003	.3712100000 +003	.37121 +003
-371.21	-.37 +003	-.3712 +003	-.3712100000 +003	-.37121 +003

LITERALS

It is often necessary to print headings, labels, and lines of text along with program results. The printing of such literals is much the same as the printing of computed variables except that any information enclosed within double less than, greater than signs is printed verbatim.

Example of literals:

```
,{<< THIS | IS | A | LINE | OF | TEXT >> },
```

All NELIAC-N characters except the absolute sign, the Boolean or, the Boolean and, and the greater than sign can be printed literally. Notice that the absolute sign | and or sign ∪ are again the necessary symbols used to indicate any spacing between words. Of course, text and variables may be intermingled within a line of print as long as care is taken to enclose the text material within the necessary double signs. Consider the following example:

```
,{ << MAXIMUM | VALUE | IS |
     EQUAL | TO > | MAXG > },
```

The variable is MAXG and is, therefore, not enclosed by the double print symbols while the literals MAXIMUM VALUE IS EQUAL TO are surrounded by the double signs.

Now suppose the variable must appear somewhere in the middle of a line of text. The following format is necessary:

```
,{<<USING| >Z< |MINES|AND|> X < |MINESWEEPERS>>},
```

Z and X are the variables and are distinguished from the literals by breaking the sense of the double print symbols.

Provisions are made to indicate the beginning of new pages, blank lines, and completion of output. These are indicated by the use of the following punctuation within a print statement but external to either single (< >) or double quotes (<< >>).

; Start new page.
, Insert blank line.
. End of file.

A statement simply to Carriage Return and Top of Form (at the present time, 8 additional CR's) would be:

```
,{<>;},
```

Commas indicate blank lines. A statement of four blank lines is written as:

```
,{<>,,,},
```

In the following statement:

```
,{<<PROBLEM | NR >|X >,,},
```

the literals PROBLEM NR and the variable X are printed followed by two blank lines. After all results are printed (actually at any time outside quotes), an end of file (ignored in line printer code, a stop code in punch or flexowriter code) may be indicated.

A single line of print for line printer output should never exceed 72 characters.

It must be remembered that the double period (..) is reserved to indicate the end of the flowchart; and may only be used for that purpose. Hence, it is impossible to place successive periods within literals since they will signify the end of the flowchart to the compiler. However, successive periods may be printed literally by inserting ALGOL words between them. The ALGOL words will prevent the compiler from detecting a double period signifying the end of the flowchart, but they will be removed from the IO statement leaving only the successive periods before the IO statement is compiled.

COMPLETE OUTPUT STATEMENTS

Although the three distinct modes of outputting - page formatting, data printout, and literals - have been discussed separately, the ability to mix them freely in output statements is necessary before the programmer can print out exactly what he wants to print out. For this purpose, it is necessary not only to understand the details of each individual type of output but to have an overall picture of their usage.

In general, an output statement in the program logic is enclosed by braces with the left brace being preceded by a comma; namely,

, { IO STATEMENT }

It is necessary to think of the existence of three levels within the output statement, these three levels corresponding to the three modes of outputting discussed above. For convenience, these three modes are called levels 0, 1, and 2, corresponding to page formatting, data printout, and literals, respectively. Entrance to an output statement through the ,{ is always at level 0. Within the output statement each < raises the level by 1 while each > lowers the level by 1, subject to the proviso that the level can never fall below 0 nor rise above 2. Exit from the output statement must be at level 0. Hence, in a typical output statement, the levels would vary as shown:

```
, { .... <<.... > .... < .... > .... > < .... < .... >>....}
     0      2      1      2      1   0    1      2      0
```

It is immediately apparent that page formatting occurs at level 0, data printout at level 1, and literals at level 2, with the appropriate rules as given on the preceding pages applying at each level.

In order to properly arrange his output lines on the page, the programmer need only keep in mind one simple rule:

Within the output statement, each time the level is increased from 0, a new line of printout is started, all oscillations between levels 1 and 2 merely change the type of printout on this line, and when the level is decreased again to level 0, this line, followed by a carriage return, will be printed. Hence, the above example calls for two lines of printout.

The programmer who has a thorough knowledge of the language used within the three modes of output, should be able to output whatever he desires by simple application of the above rule.

IO PACKAGE

NELIAC-N output is printed through return jumps to the subroutines PRINTOUT, TOP OF FORM, DOWNLINE, and END OF FILE contained in the LIBRARY PACKAGE which is automatically compiled as a separate flowchart at the end of any program which has one or more output statements. Hence, these five names should not be used by the programmer.

This chapter is ended with a sample program and resulting printed output in order to illustrate the rules covering output statements discussed in this chapter. The reader will observe that the result of this program was used to generate the full-page output statement illustration ending the section on Print Variables.

```
5
OUTPUT EXAMPLE,,..
5
A = 0, B = 00, C = 00000, E = #0, F = #00000, G = #,
AA = 0.*0, BB = 0.0*0, CC = 000.00*0, GG = 25*2,
HH = 2314*-2, II = 0*0, JJ = 00000*0;
START: ,{<>; << FIXED | POINT >>,} FIVE BLANK LINES,
492 → A, PRINT 1, 32765 → A, PRINT 1, -30 → A, PRINT 1,
BLANK LINE, BLANK LINE,
{<< FLOATING | POINT >>, << ∪ TRUE | DECIMAL | POINT >>,}
FIVE BLANK LINES, 1.0 → AA, PRINT 2,
(COMMENTS: WHAT IS WRONG WITH THE STATEMENT:
   1 → AA, PRINT 2,      )
-1.0 → AA, PRINT 2, 371.21 → AA, PRINT 2,
-371.21 → AA, PRINT 2, BLANK LINE, BLANK LINE,
{<< ∪ SCIENTIFIC | NOTATION >>,} FIVE BLANK LINES,
{< || GG ∪|| HH >,} 1.0 → GG, PRINT 3, -1.0 → GG, PRINT 3,
371.21 → GG, PRINT 3, -371.21 → GG, PRINT 3, STOP.
PRINT 1: {A → B → C → E → F → G,
   {<|| A∪ | B∪ | C∪ || E∪ |||| F∪ || G >}}
PRINT 2: {AA → BB → CC, { < ∪ AA ∪∪ || BB ∪∪ || CC > },}
BLANK LINE: {,{<>}}
FIVE BLANK LINES: { FOR I = 1 (1) 5 {BLANK LINE}}
PRINT 3: {GG → HH → II → JJ,
   { < || GG ∪ || HH ∪ || II ∪ || JJ >}}
STOP: ,{ <>; .}..
5..
```

FIXED POINT

**	***	492	***	#001ec	#00000000001ec
**	***	32765	***	#07ffd	#000000007ffd
**	-30	-30	***	-#0001e	#fffffffffe2

FLOATING POINT

TRUE DECIMAL POINT

1.	1.0	1.00
-1.	-1.0	-1.00
***	****	371.21
***	****	-371.21

SCIENTIFIC NOTATION

.25 +004	.2314 +002		
.10 +001	.1000 +001	.1000000000 +001	.10000 +001
-.10 +001	-.1000 +001	-.1000000000 +001	-.10000 +001
.37 +003	.3712 +003	.3712100000 +003	.37121 +003
-.37 +003	-.3712 +003	-.3712100000 +003	-.37121 +003

XII. ADDRESSES OF NAMES

At times, it is convenient for a variable to have as its initial value the address (location) of another variable (or, in general, the address of any name). This is handled in the dimensioning statement by following the name of the variable being defined with an equals sign and a set of braces enclosing the name of the variable whose address is to be the starting value. Of course, the variable (or name) whose name is enclosed by the braces must be defined elsewhere in the dimensioning statement or program.

Example: To define the variable ADRC and give to it as its initial value the address of the name C, the dimensioning statement must contain:

ADRC = { C } ,

A table of addresses may also be defined in the dimensioning statement, for example:

J TABLE = { P, Q, R, S } ,

J TABLE [0] contains the address of the routine P, and successive locations contain the addresses of the routines Q through S.

ABSOLUTE ADDRESSES

As discussed in Chapter II, the choice of address assignment for a variable is normally left to the compiler. However, one may choose the location of a variable in the following manner:

A = {#3ac5} ,

As a result of this assignment, the address of variable A becomes #3ac5. Obviously, A may be treated as a table consisting of consecutive locations #3ac5, #3ac6, etc. The number assigned as the address must be either a decimal or hexadecimal integer.

The mode of a variable defined in this manner is determined by placing either a comma or a period after the right brace, a comma assigning a fixed point mode to the variable and a period assigning a floating point mode to the variable. The variable A may be defined in the floating point mode as follows:

A = {#3ac5}.

Since the compiler does not take this assignment of absolute addresses into account in the compilation of the rest of the program, it should be used only for assigning addresses outside of the range of the compiled object program. In addition it should never be used for the assignment of absolute address zero.

Another NELIAC feature similar to the one just discussed, but applicable to the program logic rather than the dimensioning statement, refers to the contents of a particular address rather than the address itself. This is accomplished by using a subscript alone without reference to a named variable. This use of the subscript in the program logic will then refer directly to the corresponding absolute address in the memory of the computer itself. The following examples should clarify this point.

NELIAC STATEMENT	NOTES
, [2] → A,	The contents of memory location 2 is stored into the variable A.
, [I] → A,	The contents of the memory location whose address is in I is stored into the variable A.
, [B + 10] → A,	The contents of the memory location whose address is 10 greater than the address that is in B is stored into the variable A.
, A → [#7b5],	The value contained by the variable A is stored into memory location #7b5.
,[B]+[2] → [B+10].	The contents of the memory location whose address is in B plus the contents of memory location 2 is stored into the cell whose address is 10 greater than the address that is in B.

This form of absolute addressing is merely a degenerative form of subscripting following logically from the general form OPERAND [SUBSCRIPT ± number] where OPERAND is suppressed.

It must be remembered that absolute addresses are denoted by braces in the dimensioning statement and by brackets in the program logic.

XIII. LIBRARY OF FUNCTIONS

In scientific computation, any but the simplest problems usually require the ready availability of mathematical functions such as the trigonometric, inverse trigonometric, logarithmic, exponential, etc, functions. NELIAC-N provides these functions through its Library of Functions which, in April 1963, contains the following 14 functions:

ARCCOS
ARCSIN
ARCTAN
COS
FL TO FX
FX TO FL
EXP
LN
LOG
SIN
SPLIT
SQRT
TAN
COMSIN

The function library, whenever one or more functions are called in a program, will automatically be compiled as a separate flowchart labelled LIBRARY PACKAGE at the end of compilation just as the IO PACKAGE has been compiled. If both packages are needed in a program, the two additional flowcharts, LIBRARY PACKAGE and IO PACKAGE, will be compiled in that order at the end of compilation.

The LIBRARY PACKAGE will contain only those functions which are called in the program (and any additional functions which may be called by these functions) and not the entire function library (unless all of the library functions are called on). Hence, the length of the LIBRARY PACKAGE in any program will be the same length as if only the functions needed had been read in directly from tape.

The library function names are not forbidden names. These names may be defined and used in any program. Any library function name which is defined in a program will be used as that definition. However, if a library function name is used but not defined prior to end of compilation, this function will be compiled from the library at the end of compilation. The usual concept of temporary or local names is applicable here; namely, if a library function name is defined locally within a flowchart, that definition will be used within that flowchart but calls for that name outside that flowchart will be filled from the function library.

All functions (except FX TO FL and FL TO FX) are floating point functions. The entry in all cases is ,FUNCTION (A;B), except for SPLIT which is ,SPLIT (A;B,C),.

All arguments are floating point except the input argument to FX TO FL and the output argument to FL TO FX. FX TO FL converts a fixed point argument to its corresponding floating point value while FL TO FX converts a floating point argument to its corresponding rounded fixed point value. SPLIT converts a floating point argument into its integral and fractional parts (the output arguments appearing in that order). COMS1N is a function used by SIN, COS, and TAN for their actual computations although it may be used directly by the programmer. The input parameters of the trigonometric functions and the output parameters of the inverse trigonometric functions are in radians, and the latter are the principal values of the particular functions. The uses of all other functions should be evident from their names.

Other functions will be added to the NELIAC-N library as the demand for them arises.

As an example (much more complicated than the usual case) of the use of the library, suppose that it is required to calculate the value Y where

$$Y = \sqrt{\sin^2 (e^{2x} - \cos x) + \ln (z^2 + 3)} + 16.74$$

Dimensioning TS and TS 1 as temporary floating point working locations, a solution using the Library of Functions, is

```
,2.0 * X → TS, EXP (TS; TS),
COS (X; TS 1), TS - TS 1 → TS,
SIN (TS; TS), TS * TS → TS,
Z * Z + 3.0 → TS 1, LN (TS 1; TS 1),
TS + TS 1 → TS, SQRT (TS; TS),
TS + 16.74 → Y,
```

The general exponential $X = A^B$, where A and B are any calculable expressions, can be solved since $A^B = e^{B * \ln A}$; and, therefore,

```
,LN (A; TS), B * LN A → TS, EXP (TS; X),
```

would yield the NELIAC-N solution.

LIBRARY PACKAGE

Although library function names are not forbidden names, it is good practice to avoid using them except as library calls since their use for other purposes may complicate understanding of the program and may interfere with its integration with other flowcharts or programs. The usage of these names is further complicated by the fact that some library functions themselves call other library functions.

Hence, when the programmer uses a library function name for some other purpose, trouble may result even though he does not call that particular library function since some library function he does call may do so. Furthermore, the name of the function library LIBRARY PACKAGE should not be defined globally. At the present time, the library names with the other library functions they call indicate beneath them are:

```
LIBRARY PACKAGE
ARCCOS
   ARCTAN
   SQRT
ARCSIN
   ARCTAN
   SQRT
ARCTAN
COS
   COMSIN
   SPLIT
FL TO FX
FX TO FL
EXP
   SPLIT
LN
LOG
   LN
SIN
   COMSIN
   SPLIT
SPLIT
SQRT
TAN
   COS
   SIN
COMSIN
```

Note that several of the functions call on other functions which in turn themselves call on still other functions thus further complicating the difficulties which may arise from the indiscriminate use of library names.

XIV. MACHINE LANGUAGE CODING

The NELIAC compiler provides for the insertion of actual machine language instructions between conventional NELIAC statements by means of machine language coding also known as "crutch coding". Each instruction consists of an address -- either an unsigned decimal or hexadecimal integer or a name (which may be subscripted including the absolute address notation, but which may not be bit-handled), followed by the hexi sign and a two digit hexadecimal order (actually, any unsigned one or two digit hexadecimal number). Each such instruction is considered a statement and must be separated by commas (or their equivalent).

INSTRUCTION	NOTES
,#1f7a#50,	Load accumulator with contents of location #1f7a.
,0 #54,	Add contents of location 0.
,[I + #2000]#42,	Store result in address #2000 plus contents of index register (I).

Names of locations containing variables may be referenced as well as actual addresses.

INSTRUCTION	NOTES
,NUMBER #50,	Load accumulator with contents of location referenced by the name NUMBER.
,ALPHA #54,	Add contents of location referenced by ALPHA.
,RESULT [I] #42,	Store accumulator in location referenced by RESULT augmented by index register 1(I).

Constants may appear as address portions of many instructions. If a constant is to be treated as a hexadecimal integer, a hexi sign must precede the number.

Any statement may be labeled by the familiar method of punctuation, unique name, colon. This causes the next instruction to be compiled into a left (upper) half-word position with an appropriate right (lower) half-word pass instruction being compiled into the preceding program step if necessary. Note in the example, the conditional jump in the statement to the instruction tagged as ROUTINE.

```
          ,ALPHA [INDEX-#300]#50, [K+7]#55,
          MASK #26, 0#40,
          [LOCATION -2] #42, ROUTINE #12,
          ROUTINE [4] #11,
ROUTINE:  LOCATION #83, #1000#20,
```

There is a one-to-one correspondence between NELIAC machine-language instructions and the actual machine-language instructions in the resulting object program (allowing for "passes" caused by verbal definitions) except in the case of any instruction whose address portion contains subscripting by name.

In the pure NELIAC language, the programmer need not concern himself with the contents of the computer registers since he has no direct access to them. The compiler itself keeps track of the registers it uses thereby preventing difficulties from arising in the compiled object program due to erroneous use of the registers. However, in machine language coding, the programmer now has direct access to the NAREC registers; and, therefore, he must be careful to keep track of their contents himself. In order to be able to successfully keep track of the A and U registers of the NAREC during machine language coding, he must realize which NELIAC-N statements may destroy the register contents and avoid using any of these NELIAC statements at a time when he is interested in the contents of a NAREC register. These NELIAC-N statements include:

(1) subroutine and function calls;

(2) subscripting by name (destroys U register only);

(3) the entry or recycle test in loop control (destroys A register only);

(4) comparison statements (but not the alternatives themselves);

(5) output statements;

(6) partial word or bit handling (whether explicitly in the program logic or through dimensioned partial words);

(7) NELIAC arithmetic statements.

Examples of illegal machine language coding are:

```
(COMMENT:  ILLEGAL USE OF REGISTERS
IN MACHINE LANGUAGE CODING.)
NOUN #50, SUBROUTINE, HOLD #42,
NAME [J] #24, 0 #90,
LIST #24, STORE [E-7] #43,
NO #50, I = 0(1)6 {6#30, 0#90}
CONST #50, A = B: C#42; D#42; E→F,
AB #24, {< C >}, DE #43,
PW (5→10) #24, 0#90,
A #50, B - C → D, E #42,
```

The programmer must be particularly careful to precede the order by a hexi sign in all cases. #3000#10, not #300010, compiles as an unconditional transfer to the left instruction of location #3000. #300010, will give a compiler fault.

XV. PARALLEL NAMES

NELIAC-N provides for the parallel definition of all forms of names which may be defined either in the dimensioning statement or in the program logic. This means that whenever a name is defined, any number of additional names may be defined to have the same meaning; all of the names being completely interchangeable in their use. In all cases, except in the definition of partial words which inherently contains its own means of parallel definitions, names are defined in parallel to the initial name by simply inserting immediately after it a colon and the second name. This process of "colon name" may be repeated indefinitely, thereby defining any number of names in parallel. Whatever would have followed the single name now follows the last "colon name" in the parallel definition. Examples in the dimensioning statement

```
A : B : C,
D : E : F : G.
A1 : A2 = 57.185,
B1 : B2 : B3 (20).
```

Examples of parallel definitions in the program logic are:

,CALCULATION : REENTRY : A + B → C,...

,SUBR : SUBR1 : {0 → D → E → F}

Since any number and arrangement of partial words may be defined in parallel, the definition of identical partial words in parallel is merely the special case where both bit designations of two or more partial words are identical; e.g.,

,A : B {C (5→7), D(5→7), E(6→18)},

In this case, the names C and D are interchangeable throughout the program.

In any parallel definition, any name or names may be temporized independently of the other names in the parallel definition.

XVI. DIAGNOSTICS AND DUMPS

An effective aid for program checkout is provided by the NELIAC-N diagnostics and dumps. As an illustration, the following (nonsense) NELIAC program was compiled. The RUN INFORMATION which is automatically furnished at the end of compilation, the alphabetically sorted NAME LIST DUMP, and the OBJECT PROGRAM DUMP, either or both of the latter being optional with the compilation, were printed out. The result is also shown below.

```
5
NELIAC PROGRAM,,..
5
X(50) = 4.5*0, -29.7*0, 48.927*1, 2.0*-1,
NUMBER OF ENTRIES = 4, T = 0*0, A, B, C,
TAB: {XX(1→7), YY(20→46)} (100), ZZ;
START: A + B → C, 0 → T → N,
FOR I = 0(1) NUMBER OF ENTRIES - 1 {X[I] * X[I] + T → T}
98.7 < T ≤ X [2]:
   STOP.
   N+1 → N;
XX * ZZ (5→10) → YY,
STOP:..
5..
```

NELIACPROGRAM

NR	ROUTINE NAME	FIRST	LAST
01	START	2700	28b7

NELIACPROGRAM

NAME LIST DUMP

A	274f	
B	2750	
C	2751	
DIVIDE	28ab	
FIADD	27f6	
FIDIV	2824	
FIMUL	2806	
FISUB	27fe	
I	2701	
J	2702	
K	2703	
L	2704	
M	2705	
MULTIPLY	2898	
N	2706	
NUMBEROFENTRIES	274d	
START	27b7	
STOP	27d8	
SUBSCRIP	27e0	
T	274e	
TAB	2752	
X	271b	
XX	2752	01→07
YY	2752	20→46
ZZ	27b6	

NELIACPROGRAM

OBJECT PROGRAM DUMP

2700	27b7	10	2701	10
271b	0839	00	0000	00
271c	f7a1	26	6666	66
271d	089f	4a	28f5	c2
271e	07ec	cc	cccc	cd
274d	0000	00	0000	04
27b7	2750	50	274f	54
27b8	2751	42	27dc	50
27b9	274e	42	2706	42
27ba	27dc	50	27bc	10
27bb	2701	50	27df	54
27bc	2701	42	274d	50
27bd	27df	55	2701	55
27bc	27bf	12	27c6	10
27bf	27bf	11	27e0	10
27c0	2701	00	271b	00
27c1	0000	50	27e0	10
27c2	2701	00	271b	00
27c3	0000	24	2806	10
27c4	274e	24	27f6	10
27c5	274e	42	27bb	10
27c6	27da	50	274e	24
27c7	27c7	11	27fe	10
27c8	27cd	12	274e	50
27c9	2707	42	271d	50
27ca	2707	24	27fe	10
27cb	27cc	12	27cd	10
27cc	27cc	11	27d8	10
27cd	27df	50	2706	54
27ce	2706	42	27cf	10
27cf	27b6	50	0005	39
27d0	27de	26	0000	40
27d1	2708	42	2752	50
27d2	0001	39	27dd	26
27d3	2708	50	2898	10
27d4	27d9	26	0014	34
27d5	2709	43	2752	50
27d6	27db	26	0000	40
27d7	2709	54	2752	42
27d8	2700	82	27d9	10

27d9	0000	07	ffff	ff
27da	087c	56	6666	66
27db	8000	00	0fff	ff

27dd	0000	00	0000	7f
27de	0000	00	0000	3f
27df	0000	00	0000	01
27e0	0000	34	27eb	42
27e1	0000	44	27e4	20
27e2	27e5	20	27e9	20
27e3	27e9	22	27ea	20
27e4	0000	50	27e7	20
27e5	0000	24	0015	34
27e6	0000	40	001d	31
27e7	0000	54	0000	41
27e8	0020	34	0000	40
27e9	0000	20	27eb	50
27ea	0000	10	0000	00

27ec	27e0	82	27ed	10

27f5	0fff	ff	ffff	ff
27f6	27ef	42	27f0	43
27f7	0000	44	2888	21
27f8	27f8	11	2842	10
27f9	27f1	50	27f2	24
27fa	27fc	18	27dc	50
27fb	27f3	42	27fd	10
27fc	27df	50	27f3	42
27fd	27fd	11	2859	10
27fe	27ef	42	27f0	43
27ff	0000	44	2888	21
2800	2800	11	2842	10
2801	27f1	50	27f2	24
2802	2804	18	27df	50
2803	27f3	42	2805	10
2804	27dc	50	27f3	42
2805	2805	11	2859	10
2806	27ef	42	27f0	43
2807	0000	44	2888	21
2808	2808	11	2842	10
2809	27ef	50	0008	30
280a	27ef	42	27f0	50
280b	0001	30	27ef	61
280c	27f5	26	27ef	43
280d	27ef	50	27dc	24
280e	280f	18	2888	10

280f	27ef	50	288c	55
2810	2811	12	2813	10
2811	27ef	50	0001	31
2812	27ef	42	2815	10
2813	27ed	50	27df	55
2814	27ed	42	2815	10
2815	27ee	50	27ed	54
2816	288e	55	27ed	42
2817	288d	55	2818	10
2818	2819	12	281a	10
2819	2819	11	2889	10
281a	27ed	50	27dc	55
281b	281d	12	27dc	50
281c	27ef	42	2888	10
281d	27ed	50	0024	38
281e	27ef	54	27ef	42
281f	27f1	50	27f2	24
2820	2821	18	2823	10
2821	27dc	50	27ef	55
2822	27ef	42	2823	10
2823	2823	11	2888	10
2824	27ef	42	27f0	43
2825	0000	44	2888	21
2826	2826	11	2842	10
2827	27f0	50	0008	30
2828	27f0	42	27ef	50
2829	0001	30	27f0	70
282a	27f5	26	27ef	43
282b	27ef	50	288b	55
282c	282d	12	2831	10
282d	27ef	50	0002	31
282e	27ef	42	27df	50
282f	27ed	54	27ed	42
2830	2833	10	2831	10
2831	27ef	50	0001	31
2832	27ef	42	2833	10
2833	27ed	50	288e	54
2834	27ee	55	27ed	42
2835	288d	55	2836	10
2836	2837	12	2838	10
2837	2837	11	2889	10
2838	27ed	50	27dc	55
2839	283b	12	27dc	50
283a	27ef	42	2888	10
283b	27ed	50	0024	38
283c	27ef	54	27ef	42
283d	27f1	50	27f2	24
283e	283f	18	2841	10
283f	27dc	50	27ef	55

2840	27ef	42	2841	10
2841	2841	11	2888	10
2842	0000	44	2858	21
2843	27ef	50	27dc	55
2844	2848	12	27df	50
2845	27f1	42	27dc	50
2846	27ef	55	27ef	42
2847	2849	10	2848	10
2848	27dc	50	27f1	42
2849	27f0	50	27dc	55
284a	284e	12	27df	50
284b	27f2	42	27dc	50
284c	27f0	55	27f0	42
284d	284f	10	284e	10
284e	27dc	50	27f2	42
284f	27ef	50	0024	39
2850	2891	26	0000	40
2851	27ed	42	27f0	50
2852	0024	39	2891	26
2853	0000	40	27ee	42
2854	27ef	50	288f	26
2855	0000	40	27ef	42
2856	27f0	50	288f	26
2857	0000	40	27f0	42
2858	2858	11	0000	10
2859	27ed	50	27ee	55
285a	27f4	42	27dc	55
285b	285c	12	2860	10
285c	27f4	50	0020	38
285d	285e	20	27f0	50
285e	0000	31	27f0	42
285f	2865	10	2860	10
2860	27ee	50	27ed	42
2861	27dc	50	27f4	55
2862	0020	38	2864	20
2863	27ef	50	2864	10
2864	0000	31	27ef	42
2865	27f3	50	27dc	24
2866	2871	18	27f0	50
2867	27ef	54	27ef	42
2868	288c	55	2869	10
2869	286a	12	2870	10
286a	27ef	50	0001	31
286b	27ef	42	27df	50
286c	27ed	54	27ed	42
286d	288d	55	286e	10
286e	286f	12	2870	10
286f	286f	11	2889	10

2870	2882	10	2871	10
2871	27ef	50	27f0	55
2872	2873	12	2877	10
2873	27ef	50	27f0	55
2874	27ef	42	27dc	24
2875	2876	18	2888	10
2876	287a	10	2877	10
2877	27f0	50	27ef	55
2878	27ef	42	27df	50
2879	27f1	55	27f1	42
287a	27ef	50	2890	55
287b	287f	12	27ed	50
287c	27df	55	27ed	42
287d	27ef	50	0001	30
287e	27ef	42	287a	10
287f	27ed	50	27dc	55
2880	2882	12	27dc	50
2881	27ef	42	2888	10
2882	27ed	50	0024	38
2883	27ef	54	27ef	42
2884	27f1	50	27dc	24
2885	2886	18	2888	10
2886	27dc	50	27ef	55
2887	27ef	42	2888	10
2888	27ef	50	0000	10
2889	ffff	ff	ffff	ff
288a	27ed	82	288b	10
288b	0020	00	0000	00
288c	0010	00	0000	00
288d	0000	00	0001	00
288e	0000	00	0000	80
288f	000f	ff	ffff	ff
2890	0008	00	0000	00
2891	0000	00	0000	ff
2892	f000	00	0000	00
2893	0fff	ff	ffff	ff

2898	2894	42	2895	43
2899	0000	44	289f	21
289a	2894	50	2895	60
289b	289e	12	0000	40
289c	2893	26	0000	40
289d	2892	54	289f	11
289e	0000	40	2893	26
289f	0000	40	0000	10

28a0	2894	42	2895	43
28a1	002c	44	28b6	20
28a2	2895	53	28b6	13
28a3	2894	52	2895	57
28a4	28a5	13	0000	80
28a5	28b6	10	2894	50
28a6	2895	24	28a8	16
28a7	2893	50	28a8	11
28a8	2893	51	2897	42
28a9	2894	52	2894	42
28aa	2895	52	2895	42
28ab	0000	80	2896	42
28ac	2896	22	2895	50
28ad	0001	30	2895	42
28ae	2894	50	2895	55
28af	28ac	12	28a1	50
28b0	2896	55	28b2	20
28b1	2894	50	2895	70
28b2	0000	31	2893	26
28b3	2897	50	28b5	13
28b4	2897	43	2897	51
28b5	28b6	10	0000	40
28b6	0000	10	dddd	82
28b7	2892	82	28b8	10

The object program dump illustrated is a non-reloadable dump for information only. NELIAC-N furnishes two reloadable dumps - a bioctal dump and a standard NAREC dump. Inasmuch as the bioctal dump is approximately 40 percent as long as the NAREC dump, is comparison-loaded for correctness as soon as it is punched out, and, on readin, automatically sets its own first and last addresses and check sums itself, it is the preferred reloadable dump. In addition NELIAC-N provides for the non-reloadable dumping, in the hexadecimal format of the object program dump, of any sections of memory specified by the programmer.

In the illustration just furnished, there were no compiler-detected faults. In the event there are any compiler faults, these will be printed out as detected during compilation. The next example gives the printout of the compilation of a program containing a number of errors.

NELIACPROGRAMIOW

```
01   INPUT/OUTPUT FAULT                   ,                 >
|SQUARED|=|C,|ARE:>>,<<|||A∪|||B∪C>,>,|I=0(1)9{A[I]→BUFFER 3,B[I]→BUFFE

02   DIMENSIONING ERROR          )  TS                     (
0)=-16,,,57,-118,,16,4,-7,C(10)TS(2);NELIAC CLASS PROGRAM:SUM 100 INTEGE

02   SUBSCRIPT FAULT                  ,  INTEGERSQUARED    [
D SUM,K=1(1)100{INTEGER SQUARED[K-1)+INTEGER SQUARED SUM→INTEGER SQUARED

02   CO/OPERAND/NO FAULT          [           QUARED       +
D SUM,K=1(1)100{INTEGER SQUARED[K-1)+INTEGER SQUARED SUM→INTEGER SQUARED

02   FUNCTION FAULT                   ,  SUBSCRIP          (
 OF TABLE:L=9(-1)0{SQUARE FUNCTION(A[L]:TS),SQUARE FUNCTION(B[L];TS[1]),

02   CO/OPERAND/NO FAULT              ,  TS                )
E:L=9(-1)0{SQUARE FUNCTION(A[L]:TS),SQUARE FUNCTION(B[L];TS[1]),TS+TS(1]

02   CO/OPERAND/NO FAULT          )                        ,
E:L=9(-1)0{SQUARE FUNCTION(A[L]:TS),SQUARE FUNCTION(B[L];TS[1]),TS+TS(1]

02   FUNCTION FAULT                   +  1             A   ]
TS),SQUARE FUNCTION(B[L];TS[1]),TS+TS(1]→C[L]}EXIT.SQUARE FUNCTION(INTEG

02   UNCLOSED SUBROUTINE              .  100           A9  .
INTEGER*INTEGER→INTEGER SQ E|XT:..             ;..YZT≥U}.9DA[P},#.[↑*Z<3D4
```

NR	ROUTINE NAME	FIRST	LAST
01	IOSTATEMENT	2700	27ea
02	NELIACCLASSPROGR	27eb	28cb
03	IOPACKAGE	28cc	2c19

UNDEFINED NAME LIST DUMP

BUFER6	2c1a
EXIT	2c1b
O	2c1c
BUFFER3BUFFER3	2c1d
LOO	2c1e

Following the Program Name, there occur, in order, three different types of diagnostics. First occurs the faults in the order of detection with detailed information about each fault detected being printed out in a two-line entry. The first line gives, in order, the flowchart number, the type of fault, the current operator, the operand, and the next operator, all at the time of detection of the fault. The second line gives the 72 successive characters in the symbol string in memory, centered on the point where the compiler is compiling at the detection of the fault. This enables the programmer to quickly locate the pertinent point in his program and tells him exactly what is actually in the computer memory at this point. Next occurs the Run Information which gives the same information as for an error-free compilation. Finally, there may be an Undefined Name List. This Dump lists all names which remain undefined at the end of compilation and the locations which the compiler has assigned them at the end of the program.

The NELIAC-N compiler has a provision for loading a single flowchart without its compilation.

The NELIAC-N compiler also contains a SYMBOL STRING DUMP which will print out the actual symbol string formed in the NAREC memory from any flowchart. This is frequently of use in isolating the cause of an apparent contradiction between a flowchart and the compilation resulting from it. This symbol string dump may be used in dumping the NELIAC program during its regular compilation or it may be used with the single flowchart load without compilation provision.

APPENDIX A

Summary of the NELIAC operator symbols

A. Punctuation

, — Comma: In general, a comma is used to separate names and numbers in the dimensioning statement and to separate statements that are to be performed consecutively in the program logic. In a one name statement, a comma indicates a return jump to a subroutine. The comma is also used to separate the parameters is a function call.

: — Colon: The colon has five basic meanings. In the dimensioning statement it is used when defining a partial word, with the name of that entire word preceding the colon and a left brace following the name. Using the colon after a name preceded by punctuation defines that which follows as the subroutine or the routine associated with that name, except for parallel names.

Using a colon with any comparison symbol separates the statement of the comparison from the true alternative. The colon is also used in the definition of a function and is also used to define parallel names in both the dimensioning statement and the program logic.

; Semicolon: The semicolon is used to separate the dimensioning statement from the flowchart logic. The semicolon can also be used to end the true or false alternative of a comparison. In a function call, a semicolon separates the input parameters from the output parameters.

. Period: A period is used at the end of a sequence, when control is transferred to another part of the program as specified by the word immediately preceding the period. This same symbol is used as a decimal point in numbers and to define floating-point working locations.

.. Double period: A double period indicates the end of the flowchart logic, and, consequently, the end of the flowchart.

B. Arithmetic Operators

\+ Plus sign

\- Minus sign

* Multiplication sign

/ Division sign

↑ Exponent sign or Up arrow: Indicates an exponential operation. The number to the right of the symbol expresses the power to which the base is to be raised. At present only the base 2 (arithmetic shift) or no base (logical shift) may be used.

C. Comparison Symbols

= Equal: Also used in the dimensioning statement and in loop control.

≠ Not equal

< Less than

\> Greater than

≤ Less than or equal to

≥ Greater than or equal to

D. Miscellaneous

( ) Parentheses: In the dimensioning statement, parentheses indicate the number of variables in a table. In both dimensioning statement and program logic, parentheses enclose bit specifications for operating with partial location operands. In the definition or call of a function, parentheses enclose the parameters to be used. Parentheses also enclose comments when used with the colon. They also enclose loop increments and decrements and furnish algebraic grouping in the program logic.

[ ] Brackets: Brackets are used for subscripting. The numeral or index enclosed by brackets augments the name preceding. If no name precedes the brackets, the three quantities together are treated as an operand.

{ } **Braces:** In the dimensioning statement, braces enclose the name whose address is to be the initial value of the name preceding the braces, or enclose the number which is to be the absolute address of the name preceding the braces. They also enclose definitions of part location variables. In the program logic, braces indicate loops, and enclose subroutines, functions, and output statements.

→ **Right Arrow:** Indicates that the result of the preceding operation is to be stored into the name following the arrow. Also used to help specify bit operands.

| **Absolute Sign:** Used to purge names, used in output statements to indicate one space, and used to indicate absolute values in the program logic.

∪ **Boolean OR Sign:** Used to separate parts of a compound decision. Used in output statements to indicate five spaces.

∩ **Boolean AND Sign:** Used to separate parts of a compound decision. Used in output statements to indicate no space.

<> **Less Than, Greater Than Signs:** Used in output statements for printout of variables.

<< >> **Double Less Than, Greater Than Signs:** Used in output statements for printout of literals. Also used in the dimensioning statement for literal definitions.

5

(COMMENTS: THIS FLOWCHART DATED 4 MARCH 1963
IS A DIMENSIONING STATEMENT ILLUSTRATING
THE VARIOUS FORMS OF NOUNS IN NELIAC-N.)

```
A, B(6), C(#20), D = 5, E = -5, F = #300, G = -#f3c,
H(3) = 1, 2, 3, P(#20) = 7, 6, 5, 4,
Q(27) = , , 6, -8, #17, , 57, -#6,
R: S: T, U: V: W: X = -58, Y: Z: AA (50) = 16, -#27, , -8, #10,
AB: {AC (0→23), AD (24→47)} (26) = #1234 56 789a bc, , 5,
{AE (0→0), AF (0→7), AG (8→23), AH (0→23), AI (24→31), AJ (32→47),
   AK (24→47), AL (24→47), AM (24→47), AN (6→6), AP (15→35)},
AQ: AR: AS: {AT (5→10), AU (5→10), AV (7→14)},
AW = {#2000}, ADDR A = {A}, ADDR SWITCH = {A, B, C, D, E, F},
T|EMP, T|EMP 1: AX: {AY (5→10), T|EMP 2 (23→23)} (#10) = 57, -18,
FA. FB (6). FC (#20). FD = 5*0, FE = -5*0, FF = 278.,
FG = -768.00*0, FH (3) = 1.0, 2.0, 3.0,
FP (#20) = -12*0, -12.0, -12., -1.2*1, -12000* -3, -12.0*0, -1.2*1,
FQ (27) = , , 6*0, -8.*0, 25.0, 5700*-2, , , -6.,
FR: FS: FT. FU: FV: FW: FX = -58.0,
FY: FZ: FAA (50) = 16.0, -39*0, , -8., 16.0,
FAW = {#3000}. ADDR FA = {FA},
FADDR SWITCH = {FA, FB, FC, FD, FE, FF},
F|TEMP, F|TEMP 1: FAX (#10) = 57.0, -18.0;

NO LOGIC: ..
```

APPENDIX C

NELIAC-N Forbidden Names

NELIAC-N places the following restrictions on the programmer's otherwise unlimited choice of names which he may define and use:

(1) The 5 ALGOL words

GO TO

DO

IF

IF NOT,

FOR

must never be used as names or parts of names. However, if any of the spacing requirements are violated, the same sequence of NELIAC characters is no longer considered as an ALGOL word and may be freely used.

(2) Each name must be uniquely determined within its first 16 characters (excluding spacing and ALGOL words).

(3) The single letters I, J, K, L, M, and N must never be defined globally.

(4) The following names are defined globally in the various packages automatically compiled into programs by the compiler as needed by the programs. In many programs, some or all of them must not be used, but, in any event, good programming practice dictates that they never be used (except for the library function names and these only for bona fide library function calls):

SUBSCR1P
MULT1PLY
D1V1DE
F1ADD
F1SUB
F1MUL
F1D1V
IO PACKAGE
PR1NTOUT
TOP OF FORM
DOWNL1NE
END OF F1LE
L1BRARY PACKAGE
ARCCOS
ARCSIN
ARCTAN
COS
FL TO FX
FX TO FL
EXP
LN
LOG
SIN
SPLIT
SQRT
TAN
COMS1N

APPENDIX D

NELIAC-N CODING SHEET

CODER ____________________ DATE __________ TITLE ____________________ PROB.NO. ________ TAPE NO. ________

(:Page f________-________)

APPENDIX E

NELIAC-N OPERATOR INSTRUCTION SHEET (4/4/63) Date ______________

RCC Problem Number ______________ NRL Account Number ______________

Problem Title ______________ Programmer ______________

Sweep ______________ Telephone ______________

Console Input System Unless Otherwise Specified. If stored object program may cover Console Input System (3800-3bbb), specify Direct Operation.

Printer only (except reloadable dumps) unless otherwise specified.

1. Compile

   Flowchart Tapes:

(1) F-	(3) F-	(5) F-
(2) F-	(4) F-	(6) F-

One Tape: L0 c00.

More than one tape: L0 c01, L0 c04, L0 c03.

Stop on bad compilation unless otherwise specified. CIRCLE DUMPS DESIRED.

2. Name List Dump L0 c05.
3. OP Dump L0 c09.
4. Dump Locations (if desired):

   L0 c0a (if needed).

   Box and Transfer.

5. Bioctal Dump and Comparison Load

   Punch, L0 c06, Printer, Load Tape, L0 c07.

6. NAREC Dump

   Both, L0 c08, Printer.

7. Other Information:

   Run Information Extra Copy L0 c0d

   Printer Code L0 c0b. Punch Code L0 c0c.

8. Special Instructions: