N-63-4-3

NAVWEPS REPORT 8044
NOTS TP 3038
COPY 38

NELIAC PROGRAMMING PRIMER FOR U-490

by

Dean W. Lawrence

Underwater Ordnance Department

ABSTRACT. This report serves as an introduction to computer programming in the NELIAC language. Specifically, it is oriented toward the UNIVAC-490 digital computer, a facility of the Naval Ordnance Test Station. The language is precisely designed to translate scientific problems into coding acceptable to the machine. The resulting information is useful in the design and development of antisubmarine weapons.

U.S. NAVAL ORDNANCE TEST STATION

China Lake, California

May 1963

U. S. NAVAL ORDNANCE TEST STATION

AN ACTIVITY OF THE BUREAU OF NAVAL WEAPONS

C. BLENMAN, JR., CAPT., USN — *Commander*
WM. B. MCLEAN, PH.D. — *Technical Director*

FOREWORD

The UNIVAC-490 digital computer facility was originally installed at the Naval Ordnance Test Station to generate simulated acoustic echoes and other acoustic phenomena for a torpedo in real time.

In addition, it is also used as a general-purpose scientific tool to solve such problems as the effective size for a warhead, the strategy of weapon placement, and the solving of equations of motion.

The cooperation of the Navy Electronics Laboratory was sought in order to extend the usefulness of the UNIVAC-490 by incorporating the algorithmic language of NELIAC, the Navy Electronics Laboratory International Algol Compiler.

The computer was installed in April 1962. This report is the result of work done in Fiscal Year 1963 under Local Project 802. The report represents the considered opinions of the Torpedo Development Division and the Guidance and Control Division.

Released by
C. G. BEATTY, Head,
Torpedo Development Division
A. J. TICKNER, Head,
Guidance and Control Division
12 April 1963

Under authority of
D. J. WILCOX, Head,
Underwater Ordnance
Department

NOTS Technical Publication 3038
NAVWEPS Report 8044

Published by .... Underwater Ordnance Department
Manuscript P80/MS-86
Collation ....Cover, 22 leaves, abstract cards
First printing ....270 numbered copies
Security classification ...................UNCLASSIFIED

CONTENTS

ACKNOWLEDGMENT

The author is indebted to A. E. Lemay, LTJG, USNR, of the Navy Electronics Laboratory, San Diego, for his suggestions and corrections to the original manuscript.

INTRODUCTION

To write a program for a digital computer, the programmer must be able to communicate with the machine. This communication must be through numbers—the only language the machine can understand. Since the programmer has probably seldom used numbers as a means of communication, a serious handicap exists between man and his machine.

The first attempt to remedy this problem was the development of assembly programs, which are still widely used. These allowed the programmer to designate cells of storage and give the computer its operations in mnemonic terms.

However, the programmer still faced the task of serially programming each of the computer's various operations. For this reason, compilers were developed to accept the language of the program and translate it into numbers comprehensible to the machine. NELIAC, the compiler that is available on the UNIVAC 490, is one of many such languages. This report describes the rules, methods, and techniques of programming NELIAC. It is primarily a guide to the learning of the language, although it is hoped that it will serve as an adequate reference for NELIAC programmers.

The name NELIAC stands for Navy Electronics Laboratory International Algol Compiler.[1] The Algol language concept was formed about 5 years ago. Its primary purpose was to standardize the coding of scientific problems on computers and thereby eliminate all the existing languages and dialects that were common to only a few computers. It was to be a language, oriented to the solution of problems by numerical algorithms, that would be acceptable to any and all scientific computers.

What has since happened to Algol, as the language was later named, is academic. The various users' groups and committees still have not specified the language completely, nor have they eliminated all the ambiguities. For this reason, it was necessary for NEL personnel, under the direction of Dr. Halstead, to form their own dialect of the language when they were given the task of preparing a compiler for the Navy Tactical Data System (NTDS) service test computer. Today the language is representative of the state of the art of Algol.

[1] A complete discussion of NELIAC compilers can be found in Machine-Independent Computer Programming, by Maurice H. Halstead, published by Spartan Books, Washington, 1962.

The foremost reason for the selection of NELIAC for the NOTS computer is that the UNIVAC-490 (U-490) is a refined version of the NTDS service test computer. Hence, a great deal of the work was eliminated by the selection of NELIAC.

CONCEPTS OF NELIAC

An algorithmic compiler is by definition one that translates a numerical algorithm into the language of the machine. When a programmer has solved a problem mathematically (Step 1) he needs only to develop a computational algorithm (Step 2), record it on a paper tape or a card punch (Step 3), and load it into the machine to get a solution (Step 4). This may seem to be an oversimplification to the experienced programmer, but in reality it closely parallels the exact coding process. Although there are certain restrictions to the language and forms to be followed, they are problem-oriented.

Of the four steps, only the second requires a knowledge of NELIAC. The rest of this report, therefore, concerns only the methodology of the language.

DEFINITION OF TERMS

A flowchart is the source language program. It may vary in nature from the handling of the entire problem to doing absolutely nothing in terms of machine operations. There is no restriction on how many flowcharts may comprise the program.

The term noun refers to the name of a quantity, either variable or constant, such as --eigenvalue of matrix A-- or --coefficient of friction--, while verb denotes the name of a routine or a subroutine performing some operation, e.g., --sin-- or --find eigenvalue of matrix A-- .[2] Neither nouns nor verbs need to be defined before they are used, with some exceptions, defined under Name List, below, so long as their definition exists in some flowchart.

An operator is any one of the 25 allowable NELIAC characters that are neither numeric nor alphabetic.

The flowchart can be generalized in the following manner:

[2] Double dashes denote variable names used in examples.

```
carriage return
lower case
5
carriage return
DIMENSION STATEMENT (may be omitted)
;
name of the routine: (may be omitted)
FLOWCHART LOGIC (may be omitted)
.. (stop code)
```

The "5" signals the compiler that the flowchart is to be compiled. (It must be remembered that the "5" can perform additional functions with regard to flowcharts.)

The DIMENSION STATEMENT is the definition of all nouns used in the flowchart logic that have not been or will not be defined in another flowchart.

The semicolon defines the end of the dimension statement and the beginning of the flowchart logic. Thus, if no dimension statement were necessary in a particular flowchart, it would still be necessary to precede the logic with a semicolon. Similarly, if no logic were necessary, a semicolon would still have to follow the dimension statement.

The name of the routine is the definition of a verb. The routine must perform some sort of operation and even if it is not necessary to refer to this operation from another flowchart, it should be given a name. Otherwise, it may be omitted.

FLOWCHART LOGIC is the algorithm itself.

Regardless of the exclusion of a dimension statement or flowchart logic, the paper tape must physically end with "..(stop code)". There must be only two periods and a stop code, which is a special punch on the tape.

CO-NO TABLE

The name of each noun is assigned a unique location in memory. When this noun is operated upon in one way or another, the contents of this cell will be the numbers involved in the operation. For obvious reasons, nouns are often referred to as operands.

The symbols signifying operations are the 25 special operator characters. Although these symbols do not have a unique interpretation when considered individually (e.g., the equal symbol has two distinctly different meanings), they form a pairwise uniqueness to the compiler that is the essence of the concept of a Current-Operator - Next-Operator (CO-NO) table. For a simple explanation, the compiler reads a current operator, an operand (noun), and a next operator. This CO-NO combination tells the compiler to set up the specific CO-NO function on the operand. Once this is done, the CO and the operand are discarded, the NO becomes the new CO, and a new operand and NO are then read into the translator.

A pictorial representation of the CO-NO table and its function may be found in Appendix A. As operator techniques are developed in later sections, this Appendix will become more and more significant to those who wish to answer their own questions concerning the legality and necessity of certain CO-NO combinations.

The following example demonstrates the compiler's use of the CO-NO table.

EXAMPLE (The Symbol → Means "Placed Into.")

```
                 Flowchart   CR
                             LC
                             5
                             CR
                             X, GZZORK
                             ;
                             STORE EXAMPLE:
                             x → gzzork, ..(stop code)
```

STEP	CO	OPERAND	NO	FUNCTION
1	(CR)	x	,	Select a cell for the value of x.
2	,	GZZORK	;	Select a cell for the value of GZZORK and switch to compiler flow-chart logic.
3	;	STORE EXAMPLE	:	Select a starting cell for the routine STORE EXAMPLE.
4	:	x	→	Put x in the Q register.
5	→	GZZORK	,	Store the Q in the location of GZZORK.
6	,	____	.	Do nothing.
7	.	____	.	End.

Note that the compiler does not distinguish upper-case from lower-case letters and that the comma preceding the double period is of no consequence since it forms a do-nothing function. This example is purely demonstrative and should not be taken as the precise method of the compiler.

LEGAL NAMES AND NUMBERS

The name of any noun or verb must begin with a letter that may be followed by any combination of letters and/or numbers. The compiler associates the index registers B1, B2, B3, B4, B5, and B6 with the letters i, j, k, l, m, and n, respectively. To use index register B2, the programmer should use the name --j--. It is, therefore, illegal to dimension a variable named "i" or "j", etc.

The name length may be as long as the programmer desires, although the first 15 characters (excluding spaces) must be unique.

In summary, the three cardinal rules of naming nouns and verbs are

1. The first character must be a letter.

2. The letters i, j, k, l, m, and n may not be used as a complete name.

3. The first 15 characters, excluding spaces, must be unique, even though the name can be much longer.

Examples of Legal and Illegal Names

Legal	Illegal
A very very long name	i
A	f(t)
Z96428731B	Floating-point number
ij	

Examples of Legal Numbers

Numbers may be used in one of two modes.

Fixed Point	Example
Octal	777_8
Decimal	12345980

Floating Point	Example
Normal	3.21
Exponential	4 × 1 or 62.3 × 19

Since each fixed-point number is composed of 29 bits plus a sign, they must be less than 2^{29}-1 (or 536870911) in magnitude.

Floating point numbers may be of any magnitude the programmer desires. However, the significant digits of the number must not exceed 8. These numbers have two storage locations allocated to them: one for the characteristic (29 bits + sign), the other for the mantissa (28 bits, overflow, and sign).

Examples of Legal and Illegal Numbers

Legal	Reasons
1. 3 × -1	Same as 0.3
2. -33_8	
3. -985.23 × -9	

Illegal	Reasons
1. 12345671234567_8	$> 2^{29}$ -1
2. .231	No leading digit

Examples of Legal and Illegal Numbers (Contd.)

Illegal	Reasons
3. 3.	No decimal digit
4. 3.621_8	Floating-point octal numbers have no meaning
5. 3×10^7	10 is understood

USE OF COMMENTS

Frequently it is desirable to insert comments in the flowchart logic so that the program may be more readable to the programmer. The form for doing this is

(COMMENT: Any comment of any length)

When the flowchart is loaded into the machine, comments (and also carriage returns, upper- and lower-case punches, and spaces) are filtered out so that they cannot influence the compiled program. The programmer should, therefore, use as many (or as few) of these as he needs to make the flowchart readable.

REDUNDANT WORDS

One of the requirements specified for NELIAC is that certain words, insignificant to the compiler, be available to the programmer for insertion for readability at any point in his flowchart. These words are listed below, with the correct spacing to be used with them. It is necessary to use precisely the spacing shown (ⓑ means one blank), so that the filtering process in the compiler can remove words before compilation begins. Typical usage can be found in examples in later sections.

ⓑ if ⓑ
ⓑ do ⓑ
ⓑ if ⓑ not, ⓑ (note the comma, which must appear)
ⓑ go ⓑ to ⓑ

NAME LIST

As the compiler reads in the various flowcharts that compose some specific problem, it forms a table of names called the name list, which contains all the names of nouns and verbs, together with their starting addresses. During the process of compiling the flowcharts, the compiler refers to this list to get the addresses of the operands. To recapitulate, the process of forming a machine instruction may be thought of as looking up the CO-NO combinations in the CO-NO table (to determine the function code) and the operands in the name list (to determine the address).

The name list is complete only when the last flowchart has been compiled. Because the list contains the names of all the nouns and verbs, a name used in the logic of one flowchart may appear in the dimension statement of another. In fact, a name may be defined in only one flowchart, although it may appear in the logic of several others.

The programmer may occasionally want to use some name in one flowchart in a temporary fashion that prohibits its recognition by other flowcharts. To do this, an absolute value symbol (|) is placed anywhere after the first and before the 15th letter of the name when it is defined. In all subsequent appearances, the name should be spelled in its normal manner.

EXAMPLE

T|EMPORARY NAME (when subsequent reference is made, it would be spelled --TEMPORARY NAME--.)

When a name is temporary, it does not become an entry in the name list, and the compiler has no memory of it once it completes compilation of that particular flowchart.

There are normally two reasons for making a name temporary.

1. The name list has room for 1000_8 names; programs of sufficient size may require some names to be temporary so that there will be no overflow in the name list.

2. General-purpose subroutines (i. e., those meant for use with many different programs) require temporary names to avoid double definition. Examples of this reason may be found in the answer to Exercise 4 in Appendix B.

Names, normal or temporary, should be defined before they are used in the flowchart logic. This is not imperative, except in the case of floating point or partial words, but it does generate better machine code.

WRITING THE DIMENSION STATEMENT

As explained previously, a dimension statement is the definition of all nouns. The word definition, as used here, is meant to imply the process of assigning a block of storage (one or many cells) to each name, and placing some initial value in those cells. The initial values are loaded into these cells only when the program is physically put into the machine. It is always wise to set up initial values inside the flowchart logic when possible.

The following list of symbols are the characters that may be used in the dimension statement.

( )	=
, or .	{ }
:	[:]
{(→)}	| (temporary name symbol)

ALLOCATION OF STORAGE

( ) Definition of Length of Variable. A noun is defined when the name is listed in the dimension statement. It is assumed to have only one value unless the name is followed by a set of parentheses enclosing a fixed-point number that specifies the number of locations necessary to contain all the values of that variable, in which case the initial cell is referenced as the 0th location.

EXAMPLE 1

```
Variable name,
Multivalued variable name (20),
```

In this example, the noun --variable name-- would have one memory cell reserved for it, while --multivalued variable name-- would have 20 cells allocated.

, or . Separators. Each name in the dimension statement must be followed by a separator. The two separators are the comma or the period. Commas are generally used but not immediately after a floating-point variable name.

EXAMPLE 2

```
a, b (20), c. d (100).
```

This dimension statement would reserve one cell for --a--, 20 for --b--, one floating-point cell (actually two memory cells) for --c--, and 100 floating-point cells (200 cells in all) for --d--.

: Multiname Definition. It is often desirable to refer to a cell (or cells) by more than one name. This can be done by following the first name with a colon and the second name (similarly for a second colon and a third name, if needed).

EXAMPLE 3

```
FIRST ARRAY:  SECOND ARRAY (100).
a : B : c ,
```

This example would reserve a total of 201 memory cells. The first 200 would be set up as a 100-word, floating-point array that would be referred to as --first array-- or --second array--. The last cell would be referred to as --a--, --b--, or --c--.

{(→)} Partial Word Definition. NELIAC has the capability of performing operations on partial words. At any point in the flowchart logic, the programmer may specify a part of a previously defined noun, i.e.,

```
a (2 → 9)
```

represents bits 2 through 9 (inclusive) to be operated upon.[3] No other bit in the word will be disturbed.

[3] In the U-490, the low-order bit is 0, the high-order is 29.

This method of bit handling is cumbersome, particularly when the partial notation must be used many times in the course of the program. To alleviate this awkwardness, the compiler can recognize a name as a partial word when it has been defined as such in the dimension statement. The notation for this is

$$\{a\ (2 \rightarrow 9)\}$$

When the name --a-- is used in the logic, only bits 2 through 9 will be affected. The following example further explains the use of partial words.

EXAMPLE 4

$$A : B : \{\text{JUMP FLAG 1}\ (1 \rightarrow 1),\ \text{JUMP FLAG 0}\ (0 \rightarrow 0),\ \text{B UPPER}\ (15 \rightarrow 29),\ \text{A LOWER}\ (0 \rightarrow 14)\}\ (20),$$

In this example --JUMP FLAG 1-- is the first bit of each --A-- (or --B-- or --B UPPER--), --JUMP FLAG 0-- is the 0th bit of each --A--, --B UPPER-- is the 15 most significant bits of --A--, and --A LOWER-- designates the 15 least significant bits of each --A--. Note that only 20 cells are allocated by this dimension statement.

SPECIFICATION OF CELL CONTENTS

Up to this point, the discussion has been centered on the allocation function of the dimension statement. The remainder of this section will be devoted to the method of specifying the contents of a name in the dimension statement.

= Numerical Values. All variables will be set to zero immediately before execution of the program unless the name is followed by an equality symbol and the necessary data values. The comma is used as a separator between and following the data in all cases. The omission of one or more pieces of data simply forces zeros into the corresponding memory locations. Example 5 further illustrates these techniques.

EXAMPLE 5 (May Be Considered in the Whole)

Dimension	Comment
A.	Reserves one floating point location that initially is 0.
B(3) = 1,	Three locations reserved; the first is set to 1, the other two to 0.
$\{C(6 \rightarrow 11)\}\ (2) = 4300_8,\ 3700_8,$	Places 43_8 into the 6th to 11th bits of first cell of C and 37_8 in the same bits of the next.
E(5) = 1.2, 4 × 5,, 3.1,	The five locations (floating point) of E contain, respectively, 1.2, 400,000, 0.0, 3.1, 0.0.

Note that the first numerical value in a list determines the mode of the variable (i.e., fixed or floating), and that the period is never used as a separator in or after a list of values.

{ } Address Switches and Jump Tables. In normal scientific problems, the programmer is usually interested in the contents of a variable name and not where the variable is located in memory. However, it is occasionally necessary to have this information. An address switch is a variable (one or more cells) that contains the address of another noun in the lower half of each word in the switch.[4] The notation for this is

SWITCH 1 = {x, y, z},

Thus, --SWITCH 1-- is a variable name and is allocated three cells in memory, the first of which contains the address of the variable --x--, and successive locations contain the addresses of --y-- and --z--. An address switch may contain as many names of nouns as is desired, and is, in the case of more than one name, actually a vector of several locations even though the length is not numerically specified.

A jump table serves the same function for verbs that an address switch does for nouns, i.e., a jump table is a name defined to be equal to the address or addresses of one or more verbs.

EXAMPLE 6

JUMP TABLE 3 = {entry point 1, entry point 2},

This example specifies a noun, called --JUMP TABLE 3--, which is two cells in length, the first containing the address of the starting location of subroutine --entry point 1-- and the second, the starting location of --entry point 2--.

> Caution. Although an address switch and a jump table are identical in physical appearance, their translations are different and the two should never be mixed. That is, names of nouns and verbs should not be mixed in the argument of a jump table or an address switch.

[:] Literals. This is the only method of entering alphanumeric data.

EXAMPLE 7

[TEXT : SOME ALPHANUMERIC INFORMATION. a<B: x→a; y→b;]

The literal begins with the first character after the colon and ends immediately before the right bracket. Any NELIAC symbol may be used in the literal except the right bracket. All spaces in the literal are

[4] For the U-490, bits 18 to 20 contain the k designator appropriate to the name. By defining --SWITCH 1-- (above) as {SWITCH 1 (0→14)}, the possibility of a k-designator malfunction may be eliminated.

filtered out except those both preceded and followed by an alphanumeric character. (Filtering occurs when two or more consecutive spaces are present).

When the noun --text-- is used, the address of the literal will be obtained.

The literal is formed internally as the compiler code (Appendix C), packed five characters to a word from left to right. A full zero cell always follows a literal in memory.

WRITING THE FLOWCHART LOGIC

SUBSCRIPTING

Subscripts to variables are denoted by a value enclosed in brackets following the variable's name, e.g., A [42].

The subscript itself may be any one of the following three forms:

1. An integer (as above)
2. An index register variable (i, j, . . . , n) such as A [i]
3. An index register variable ± an integer, such as A [i - 23]

Any variable may be subscripted regardless of its definition in the dimension statement.

When the subscripting of a partial word field is required, the partial word notation follows the subscript.

EXAMPLE 8

$$A[i - 23]\ (0 \rightarrow 3)$$

If A has been previously defined as a partial word, bit 0 is the lowest ordered bit specified in the dimension statement.

Jump tables and address switches are referred to in the logic as indexed variables.

EXAMPLE 9

Table 3 [i], Switch 4 [2]

This example designates the ith entry in the jump table or address switch, named --Table 3--, and the second entry in --Switch 4-- (where the first entry in the table is actually the 0th).

By removing the variable name, subscripts can also be used to designate absolute locations in memory.

EXAMPLE 10

[i], [249], [i + 12]

This example calls out successively the contents of the cell whose address is in i, the contents of Cell 249, and the cell whose address is i + 12. Exercise 2 in Appendix B contains another example of this use of subscripts.

ARITHMETIC OPERATIONS

The six arithmetic operations and their symbols are

1. Storing ($\rightarrow$)
2. Addition (+)
3. Subtraction (-)
4. Multiplication (X)
5. Division (/)
6. Left or right shift n places ($\times 2 \uparrow n$ or $/ 2 \uparrow n$)

Arithmetic operations are always shown to the left of the store symbol. The over-all result of any computation must be stored with the exception of the arithmetic in a decision (see the section below on Decision Making).

EXAMPLE 11a

$$A / B \times C - D \rightarrow E \left(E = \frac{AC}{B} - D \right)$$

Any number of parentheses may be used to indicate algebraic grouping.

EXAMPLE 11b

$$A / (B \times C) - D \rightarrow E \left(E = \frac{A}{BC} - D \right)$$

EXAMPLE 11c

$$(A / B \times (C - D)) \rightarrow E \left(E = \frac{A}{B} (C - D) \right)$$

EXAMPLE 11d

$$(A / (B \times (C - D))) \rightarrow E \left(E = \frac{A}{B(C - D)} \right)$$

The normal rules of arithmetic precedence are strictly observed, i.e., exponentiation (shifting) is performed first, multiplications and divisions second, and additions and subtractions last. This precedence is applied successively to each group from innermost to outermost (Examples 11c, d). Computations on the same precedence level (e.g., multiplication and division) are performed from left to right (Examples 11a, b).

Intermediate results during a computation may be stored in the following manner:

EXAMPLE 12

$$(A + B \rightarrow C) / (E - F \rightarrow D) \rightarrow G$$

Example 12 is arithmetically the same as:

$$A + B \rightarrow C,\ E - F \rightarrow D,\ C/D \rightarrow G$$

Fixed-point expressions may be shifted right or left n places by the symbols $/ 2 \uparrow n$ or $\times 2 \uparrow n$. The U-490 left shift is circular, while the right shift is not, although it does employ sign extension. It is extremely important to recall these two shifting characteristics when programming the U-490.

The number of places to be shifted, n, may be a fixed-point constant, an index-register variable, or a fixed-point variable name.

Modes of arithmetic (i.e., fixed and floating) should never be mixed on the same side of the store symbol. When the two modes are separated by this symbol, floating to fixed (or vice versa) conversion is indicated.

EXAMPLE 13

(Assume that --fixed pt-- is dimensioned as fixed point and --floating name-- is a floating-point variable)

Fixed pt + K → floating name	Floating-point representation of the integer (fixed pt + K) is stored in --floating name--.
Floating name → fixed pt	The decimal fraction of --floating name-- is truncated and the integer value is placed in --fixed pt--.

LOOPS

One of the most common digital programming techniques is the iterative procedure or loop. A loop, in the NELIAC sense, is any sequence of one or more steps of the algorithm that needs to be repeated.

There are two parts to a NELIAC loop: the sequence of steps to be repeated, and the loop control, which regulates the number of times the sequence is to be repeated.

Loop control is set up on an index register by listing the particular register (i.e., i, j, k, l, m, or n) followed by an equality (=) symbol, the beginning value that the register should have, the increment (or decrement) of each step enclosed in parentheses, and the final value. The steps of the algorithm, enclosed in braces, follow the loop control.

EXAMPLE 14

i = Beginning value (increment) final value {a[i] → b[i], c → d[i]}

The execution of Example 14 can be characterized by the following steps:

1. Place --beginning value-- in i.

2. Execute the steps within the braces.

3. Check i to see if it is equal to --final value--. If it is, set i to 0, and go to the first step after the right brace; if not, add --increment-- to i and go back to Step 2.

From this characterization, several important conclusions about loops can be reached.

1. Every loop is executed at least once, since the test follows execution of the algorithm.

2. The index register must equal --final value-- to obtain a normal exit from the loop. Therefore, extreme caution should be used when --increment-- and --final value-- are chosen, to insure that an equality condition will exist at the correct time.

3. By choosing a negative value for --increment-- and making sure that --beginning value-- is larger than --final value--, a valid decrementing loop is possible.

4. Under normal exit conditions, the register used in the loop control is always set to zero. If, however, during the execution of the loop, transfer outside the loop is made, the register will contain its last value (see the section on Decision-Making, below).

Recalling the names used in Example 14, the following rules apply:

--Beginning value-- may be an integer, a fixed-point whole or half word (with subscript, if necessary), an index register, or an index register ± an integer.

--Increment-- must be an integer (with a minus sign, if decrementing is desired).

--Final value-- is the same form as --beginning value--. If it is the number zero, decrementing is automatic regardless of the sign of --increment--.

EXAMPLE 15 (Valid Loop Controls)

```
i = k (2) k + 10 {----}
j = ts[i] (1) 126 {----}
```

<u>Caution.</u> Never index across zero on the U-490,
<u>e.g., i</u> = 50 (-1) - 50

TRANSFER POINTS

At any point in the flowchart logic, the programmer may define the name of a type of verb, called a transfer point, by entering the name and following it with a colon.

EXAMPLE 16

$$a + b \rightarrow c,$$

STORE C:

$$c \rightarrow d \rightarrow e, \text{ Store c.}$$

This example would do the following:

1. Add a and b, and store the result in c
2. Store c in d and e
3. Transfer control to the first statement following the name --Store c-- (which, for this example, has set up an infinite loop, a highly undesirable program characteristic)

Considering Steps 1 and 2, of Example 16, the definition of a name obviously does not interrupt the logical flow of the algorithm. It does, however, give the programmer a method of referring to a specific point in the flow.

Note that the punctuation following the usage (not definition) of --Store c-- is a period. It acts as a comma normally does, except that it sets up an unconditional transfer to the address of the name --Store c--.

DECISION-MAKING

There are six basic decisions that NELIAC can make about two variables of a similar mode. They are

$A = B$	$A \geq B$
$A \neq B$	$A < B$
$A \leq B$	$A > B$

There is also one basic decision for fixed point variables only.

$$A < B < C \text{ (interval decision)}$$

In addition to the above basic decisions, compounded decisions may be formed by utilizing the Boolean "and" ($\cap$) and "or" ($\cup$) operators. Up to 16 simple decisions may be strung together, provided only "ands" or "ors" are used (i.e., no mixing of $\cap$ and $\cup$ is permitted).

The decision to be made is followed by a true and a false alternative. The complete punctuation format looks like:

A = B: --true alternative--; --false alternative--;

The colon indicates the end of the decision and the beginning of the true alternative (i.e., the steps that should be executed if A is equal to B). The first semicolon indicates the end of the true alternative; the second, the end of the false alternative. In no case can both the true and false alternatives be executed. If the statement of the decision (i.e., A = B) is true, the true alternative is executed and control is automatically transferred to the first step <u>after</u> the false alternative. If the statement is false, control is transferred to the false alternative (which is then executed) and the program continues from the first step after that alternative.

The period, used in the sense indicated in the section on Transfer Points, will also end an alternative. In this case, it should be recalled, control would be transferred unconditionally to the transfer point and no return to the first statement after the false alternative would be generated.

EXAMPLE 17 (Valid Decisions)

$A \leq B \cap B < C \cap C \geq D$: ; $a \rightarrow c$; continue with flow (If it is true that $A \leq B < C \geq D$, continue with flow; if not, store a in c and continue with flow.)

$X \neq Y$: transfer pt 1. ; continue with flow (If it is true that $X \neq Y$, unconditionally transfer to --transfer pt 1-- and continue the flow from there; if not, continue the flow from this point. The period indicates the end of the true alternative, and the semicolon is the end of the false portion.)

Z = GZZORK: True transfer. False transfer. (Both the true and the false alternatives may be unconditional transfers.)

Decisions may be nested (i. e., a decision in an alternative of a previous decision) as deeply as needed. For clarity, it is permissible to enclose the whole true or false alternative in a set of braces when the alternative contains at least one decision.

The following example demonstrates the nesting of decisions.

EXAMPLE 18

$A = B : Z \rightarrow$ GZZORK, Q. $\{x \neq y :$ GZZORK $\rightarrow y$; $y \rightarrow$ GZZORK; $A \rightarrow B\}$;

This may also be written as

```
A = B: Z → GZZORK, Q.
          x ≠ y: GZZORK → y; y → GZZORK; A → B;
```

Arithmetic, including algebraic grouping, and bit handling may be used on either side of a decision operator.

EXAMPLE 19

$A \times B + C \rightarrow D < E + F$: true; false;

$A \times (B + C) < D\ (0 \rightarrow 23)$: true; false;

For reference, a review of punctuation operator usage is included in Appendix D.

SUBROUTINES

When a problem is being programmed, it is often desirable to set off a section of the algorithm and designate it by a name. Then, when this section is needed, a simple listing of the name will be all that is required. These sections are called subroutines and the process of sectioning is called program partitioning. Some classic examples of processes that are usually made into subroutines are trigonometric functions and their inverses, square root, sorting routines, and matrix inverting routines.

The skeletonized format of a subroutine is shown below.

SUBROUTINE NAME: {- - - - - - - - - - -}

The colon, as in the case of the transfer point, denotes the definition of a verbal name. Braces enclose the steps of the subroutine.

There is no restriction to the number of subroutines that may be programmed in a flowchart (i. e., a flowchart may consist of a main routine and many subroutines, or it may consist of only one subroutine).

Normally, the nouns in a subroutine are entered in the dimension statement of the flowchart. However, subroutines may also have a list of nouns, enclosed in parentheses, that are common only to the subroutine. These nouns are called "dummy" variables because they are the arbitrary (or changing) input or output nouns.

EXAMPLE 20

Suppose it is desired to write a subroutine that could compute the sine of some angle, α. We would then want α to be a dummy variable, since it would be a changing parameter. This subroutine[5] would be defined in the following manner:

SIN (ALPHA): {---------}

Suppose, further, that in the flowchart logic of the main program (or another subroutine) it is desired to refer to this --SIN (ALPHA)-- routine to compute the sine of a variable named --Z--. This would be called out by --SIN (Z)--. The value of Z would be used in the computations everywhere that --ALPHA-- originally appeared in the flowchart of SIN.

As many arguments (dummy variables) may be used as is desired. However, those that are used in an output sense (i. e., those values that are developed to return to the calling program) must follow the input parameters and the two must be separated by a semicolon. At least one input argument must exist if there are output arguments.

The list of dummy variables may be punctuated exactly as a dimension statement except for the semicolon separating input and output variables.

---

[5] In previous NELIAC literature, a function was defined to be a subroutine with associated parameters (i. e., a list of dummy variables). This distinction is unnecessary, and is avoided, since the scope of this report is confined to the usage of NELIAC.

EXAMPLE 21

FUNCTION 1 (A, B, C; {D (0 → 5)}): {----------}

A, B, C are input dummy variables: partial word D is an output argument.

FUNCTION 2 ({a(0 → 5), B(10 → 20)} (20) = 7, 10, c): {-----}

Twenty-one dummy cells are reserved by this argument list. Bits 0 to 5 of the first 20 may be referred to as --A--, while bits 10 to 20 are called --B--. A [0] = 7; A [1] = 10; all other A's (and B's) are zero. The 21st cell is named --C--.

USE OF SUBROUTINES

There are two distinct ways a subroutine is used: as a noun, and as a verb.

A typical example of usage as a noun is the aforementioned --SIN (ALPHA)-- routine, since it would be used in a fashion such as:

SIN (Z) X HYPOTENUSE → OPPOSITE SIDE,

When a subroutine is used as a noun, it is fairly obvious that the computer must assume that the result of the subroutine will be in a specific arithmetic register. If the programmer is unsure of where the answer of a function is physically, he should end the function by the simple store command

, answer → answer }

This will insure that the proper values are correctly placed. The convention for the U-490 is that the answer be in the Q register.

An example of a subroutine used as a verb may be found in Exercise 2 of Appendix B. This problem requires that several blocks of storage be sorted so that the contents of each are arranged in ascending order. As a result, a subroutine that could not be used in an arithmetic statement is also required.

The concept of using a subroutine as a verb is precisely the same as the usage of transfer points (see the section above on Transfer Points), except that control is returned to the calling point. Since this requires uniqueness in the CO-NO sense, the proper punctuation following the subroutine name (in flowchart usage) is the comma (as opposed to the period for transfer-point names).

EXAMPLE 22 (Valid Subroutine Usage)

A < B: SORT, B → A, Return to Start. A → B;

In Example 22, --SORT-- is the name of a subroutine and --Return to Start-- is the name of a transfer point. If A were less than B, control would be transferred to the first location of that subroutine. When the computations of the subroutine are completed, control is transferred back to the calling point, --B-- is stored in --A--, and the control is permanently transferred to --Return to Start--.

MACHINE CODE

NELIAC can accept machine code at any point in the flowchart logic. The following are some examples of U-490 code:

10	000_8	0 ,	(Clear q register)
26	030_8	a ,	(Add the whole word --a--)
14	030_8	b [j-1] ,	(Store in the whole word --b [j-1]--)
26	030_8	16543_8 ,	(Add the whole word contained in location 16543_8)

Each machine command must begin with the five octal digits corresponding to the f, j, k, and b designators followed by the octal sign. At least one digit (or the name) of the operand must be listed, and each command should be terminated with a comma.

Subscripting of the name of an operand is permissible. If both subscripting and a non-zero b designator are written, the subscripting takes precedence.

GENERATION OF OUTPUT ON THE HIGH-SPEED PRINTER

The standard mode of output for scientific programs on the NOTS U-490 is the on-line printer. Since input-output is not a part of an ALGOL-type language (because it not only is machine-dependent but is also installation-dependent), it has generally been left to the individual programmer to do by utilizing the machine code capabilities of the language. As an attempt to improve this situation, a general-purpose subroutine named PRINT for output on the printer has been implemented.

PRINT has provisions for 31 arguments, the first of which must be the name of a literal acting as the format. The following 30 arguments should be the names of nouns to be output. These nouns are referred to as the list.

THE FORMAT

Before the computer can set up a line of print for the printer, it must be able to determine the form of the line. It does this by taking the information specified by the literal in the argument, henceforth referred to as the format.

There are 128 print positions to a line on a UNIVAC printer. The line may be thought of as composed of one or more adjacent fields. A field consists of only one type of output, e.g., a Hollerith field, or a floating-point number field. Thus, if it was desired that a line of print read

"The present value of theta is xx.xxx"
(where xx.xxx represents a variable floating-point number),

it would be broken into a Hollerith field followed by a floating-point field.

FIELD SPECIFICATION

Numerical fields are specified by the letters I, F, or E. An I field specifies a fixed-point integer to be output; F and E are designations for floating-point nouns. The difference between E and F is that an F-type outputs the number as a decimal while E-types appear as exponential forms (of ten).

The precise specification of the number fields is as follows:

Iw, Fw.d, Ew.d

where

w = field width
d = number of desired decimal places

Each I, F, or E specification must be followed by a comma.

EXAMPLE 23

```
CR
LC
5
CR
A = 12,
B(2) = 22.222222, -6666.666,
[FORMAT 1: I20, F16.3, E13.5,];
PRINT(FORMAT 1, A, B[0], B[1]), .. (stop code)
```

The preceding example would print this line:

```
 18 blanks       10 blanks                  ┌─ sign position
←——————————→ 12 ←——————————→ 22.222 -6.66666E ▾03
```

Note that the decimal point is included in w, as is the sign. In the exponential type, four extra positions for "E sign xx" must be included in the width count.

There also exists the capability of repeating a numerical field by using

nIw, or nEw.d, or nFw.d

EXAMPLE 24

Assuming

[FMAT 2:3I20,] and A(3) = 12, 13, 14

then

PRINT(FMAT 2, A[0], A[1], A[2]) would give

```
 18 blanks       18 blanks       18 blanks       all blanks
←——————————→ 12 ←——————————→ 13 ←——————————→ 14 ←——————————→
```

Alphanumeric fields are specified by the symbols

< >

which are often referred to as "quotation marks" to clarify their use. All alphanumeric characters, which are available on the high-speed printer, and are contained within the quotation marks, will be printed precisely as written except for two or more consecutive spaces. All consecutive spaces, except for the first, are omitted.

EXAMPLE 25

If the literal is

[FORMAT 3: < THIS IS ALPHANUMERIC INFORMATION. >]

and PRINT (FORMAT 3) is used, the following line results:

THIS IS ALPHANUMERIC INFORMATION. $\xleftrightarrow{\text{all blank to edge of paper}}$

Blank fields (in addition to the method shown in the section on numerical fields) may be generated as by the following symbology:

$|n|$, where $1 \leq n \leq 127$

The number n specifies the number of blank spaces to be inserted.

The end of a line of print may be signified by a solidus (/). This is normally used prior to the physical end of the format. The end of the format (regardless of the last punctuation) terminates the loading of the printing buffer, and outputs that buffer. Thus, if a slash (/) were the last character in the format, the buffer would be printed, and the end of the format would again cause the buffer to be printed, thus giving a blank line.

Blank lines can be generated by one two techniques. The first is the use of multiple slashes, such as "////". Since the slash indicates the end of a line of print, this notation would

1. Print the last line generated
2. Generate three blank lines

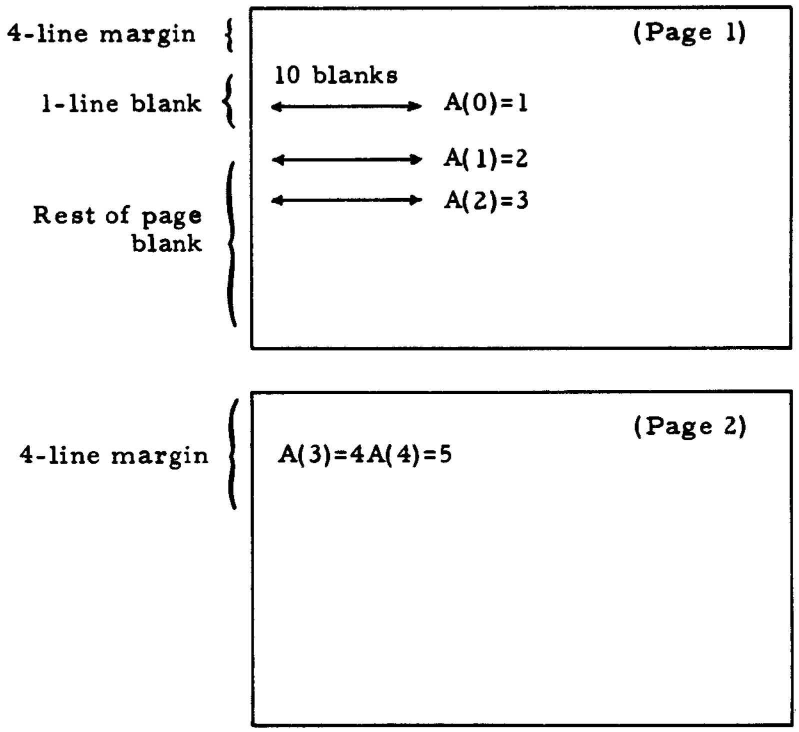
Note that if these are the first symbols in the format, they would effectively give four blank lines.

The second method of generating large quantities of space is by use of the exponentiation arrow (↑). This symbol does the same thing as the slash but, in addition, causes the printer to move up to the top of the next page.

EXAMPLE 26

```
CR
LC
5
CR
A(5) = 1, 2, 3, 4, 5,
[FMAT 4: |10| < A(0) = > I1,//|10| < A(1) = > I1,/|10|
      < A(2) = > I1, ↑ < A(3) = > I1, < A(4) = > I1,];
PRINT(FMAT 4, A[0], A[1], A[2], A[3], A[4]), .. (stop code)
```

This example would print the following:



MISCELLANEOUS INFORMATION

Since the end of the format generates a line of print, a new line will be printed the next time PRINT is used. Therefore, there is no possibility of overprinting on the previous line.

Because of the amount of time required for a print cycle, a program will be executed faster if only single lines are generated in PRINT (although blank lines require essentially no time to print). By doing this, and by placing a portion of the algorithm between the PRINT calls when possible, the programmer will maximize the buffering capability of the machine.

There are 66 possible lines of print on the paper used in the printer. PRINT will automatically leave a top and bottom margin of four lines each, leaving the programmer with 58 possible lines per page.

THE NOUN LIST

The only limitations placed on the list of names following the format name in the argument of PRINT is that they be defined nouns, and that the total number not exceed 30. However, since floating-point nouns require two addresses, the list may have to be as short as 15 nouns.

AVAILABLE GENERAL-PURPOSE SUBROUTINES

The subroutines in this section are common only to the U-490 at NOTS and are not a part of the actual NELIAC compiler.

OPEN SUBROUTINES

These routines are termed open since the machine code of each is inserted in the object (machine code) program each time they are referred to in the source program (flowchart).

These routines must be punctuated precisely as shown (except for spacing).

[PAUSE < --transfer point name-- >,],

--Pause-- will cause the machine to transfer to the --transfer point name-- and stop. When the high-speed switch is depressed, the next instruction to be executed will be located at the address associated with --transfer point name--.

[STOP <,],

--Stop-- differs from --pause-- only in the sense that no more instructions in the program may be executed once --stop-- has been encountered. This is the normal method of ending a NELIAC program.

CLOSED FUNCTIONS

The following general-purpose arithmetic functions are available to the programmer. All arguments are floating-point variables.

1. <u>SINF (ARG).</u> Computes sine, includes COSF.
2. <u>SECF (ARG)</u>. Uses COSF, computes secant.
3. <u>CSCF (ARG)</u>. Uses SINF, computes cosecant.
4. <u>LOGF (ARG)</u>. Computes natural log of ARG.
5. <u>EXPF (ARG)</u>. Computes e to ARG.
6. <u>TANF (ARG)</u>. Computes tangent.

7. COTF (ARG). Computes cotangent using TANF.
8. ASINF (ARG). Uses SQRTF and ATANF, computes arcsin between -PI/2 and +PI/2.
9. ATANF (Ordinate, Abscissa). Computes arctan between -PI and +PI.
10. ACOTF (Ordinate, Abscissa). Uses ATANF, computes arccot between -PI and +PI.
11. ABSF (ARG). Finds absolute value of floating-point variable.
12. SQRTF (ARG). Computes square root of positive floating-point variable.
13. SOLVPOLYEQ. Uses SQRTF, finds roots of a polynomial equation.
14. SOLVDE. Solves system of ordinary differential equations.
15. SLVEQ. Solves linear equations.
16. EVALMATRIX. Gives rank, determinant, and inverse of matrix and solves a system of linear equations.
17. TIME THIS SUBROUTINE. Times subroutine using real-time clock.
18. SR INTEGRATION. Simpson's rule integration.
19. LAQ INTEGRATION. Uses EXPF, integration by Gauss-Legendre guadrature.
20. LEQ INTEGRATION. Integration by Gauss-Laguerre quadrature.

Appendix A

CO-NO TABLE

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

0.	Fault	15.	Generate mult or enter
1.	Initiate relation control	16.	Generate divide
2.	Fault	17.	Generate div or enter
3.	Generate straight jump	18.	Generate div quant
4.	Generate return jump	19.	Generate store
5.	Check partial word	20.	Initiate loop control
6.	Check for algebra	21.	Set exit conditions
7.	Check for neg loop increment	22.	Generate IO
8.	Check for loop limits	23.	Initiate subscript
9.	Clear temp list	24.	Modify subscript
10.	Generate add or enter	25.	Set subscript
11.	Generate add	26.	Save current operator
12.	Generate subtract	27.	Generate add or enter
13.	Generate multiply	28.	Initiate relation control
14.	Generate mult quant	29.	Generator exit

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

Appendix B
EXERCISES

EXERCISE 1

Write a complete dimension statement that will do the following:

1. Define the fixed-point cells X1, X2, ..., X6.
2. Define the floating-point vector X[0], X[1], ..., X[5].
3. Define the 200-celled vector, B, such that you can operate on the whole word of each B[i], on the lower half of each B[i] (call it B LOWER [i]), on the upper half of each B[i] (call it B UPPER [i]), or on A[i] = B[i + 100] for i = 0, ..., 99.
4. Referring to part 3, show a method of insuring that all of the B[i] = 0 for i = 1, 2, and 4, 5, ..., 199 and B[0] = 9, B[3] = 14.
5. Define the noun LOCATION to be the address of B[0] (i.e., the number stored in LOCATION will be the address of B[0]).

EXERCISE 2

Assume that there are three arrays of 100 cells each (call them A, B, and C) that have some number in each cell. Write a routine that will sort the values in the three vectors in ascending order (do each vector separately). See Fig. 1 for a typical flow diagram.

EXERCISE 3

Assume that there is no arithmetical division hardware available for the U-490 and that we need a subroutine to accomplish this arithmetic. One way of doing this (for left-justified, positive, fixed-point numbers) is by the following algorithm:

1. Subtract the DIVISOR from the DIVIDEND.

2. If the REMAINDER ≥ 0, set qn = 1 (the nth bit of the QUOTIENT), set the DIVIDEND = REMAINDER, shift the DIVISOR one bit to the right, and go to Step 4.

3. If the REMAINDER is < 0, set qn = 0 and shift the DIVISOR one bit to the right.

4. Have you done this process K + 1 times? If so, go to Step 5. If not, go back to Step 1 after making n = n - 1.

5. If DIVIDEND - DIVISOR is > 0, add 1 to QUOTIENT and then end the routine.

In this algorithm, the initial value of n would be K, the number of times we must go through the process.

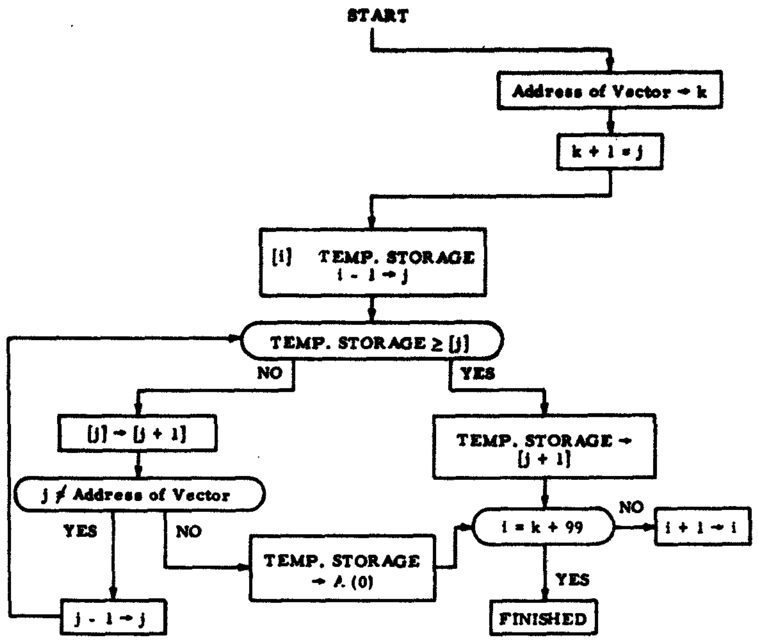


FIG. 1. Typical Flowchart for Sorting Elements of One Vector.

Assume that in K there is a value (<30), and the fixed point numbers DIVIDEND and DIVISOR that are left-justified to each other (i.e., the first bit in DIVIDEND that is a 1 is also a 1 in the corresponding position of DIVISOR and vice versa). Write a subroutine that has DIVIDEND, DIVISOR, and SIGN FLAG as input variables and QUOTIENT as an output parameter that will do the indicated division. After the division is completed, set QUOTIENT negative if SIGN FLAG is $\neq 0$.

EXERCISE 4

Hastings[6] suggests that the following approximation for

$$\sin \frac{\pi}{2} x$$

is good to eight places:

[6] Hastings, Cecil, Jr., Jeanne T. Hayward, and James P. Wong, Jr. Approximations for Digital Computers. Princeton, N.J., Princeton University Press, 1955.

$$\sin\frac{\pi}{2}x = C_1x + C_3x^3 + C_5x^5 + C_7x^7 + C_9x^9$$

where

$C_1 = 1.57079,631847$	$C_7 = -0.00467,376557$
$C_3 = -0.64596,371106$	$C_9 = 0.00015,148419$
$C_5 = 0.07968,967928$	

Write a function that will have the floating-point variable Z as its argument so that it will compute sin Z. Note that

$$x = \frac{2}{\pi} Z$$

Will the subroutine be good to eight places?

EXERCISE 5

Two positive fixed-point numbers are said to be left-justified with respect to each other when the smaller is shifted left until the first 1 bit in it is in the same position as the first 1 bit in the larger, e.g.,

LARGER = 0011010010
SMALLER = 0000100111

SMALLER requires two left shifts to be left-justified to LARGER.

Write a function that will do the following:

1. Accept as inputs the variables LARGER and SMALLER such that |SMALLER| < |LARGER|.

2. Have as an output the variable SIGN FLAG that will be 1 if, and only if, either SMALLER or LARGER is negative (but not both negative): otherwise SIGN FLAG = 0.

3. Set the inputs positive if they are negative and left-justify SMALLER to LARGER.

4. Count the number of shifts required to make the justification and have that number available in the K register when you exit from the routine.

A suggested flowchart is shown in Fig. 2.

EXERCISE 6

Assume 100 values in each of the fixed-point vectors A and B, and 100 values in the floating-point vector C. Write a program utilizing Exercises 2 to 5 as subprograms, which will do the following:

1. Sort A and B in ascending order.
2. Compare the |Ai| to |Bi| for i = 0, ..., 99. If Ai ≥ Bi, justify Bi to Ai and divide B into A using the K developed in Exercise 5

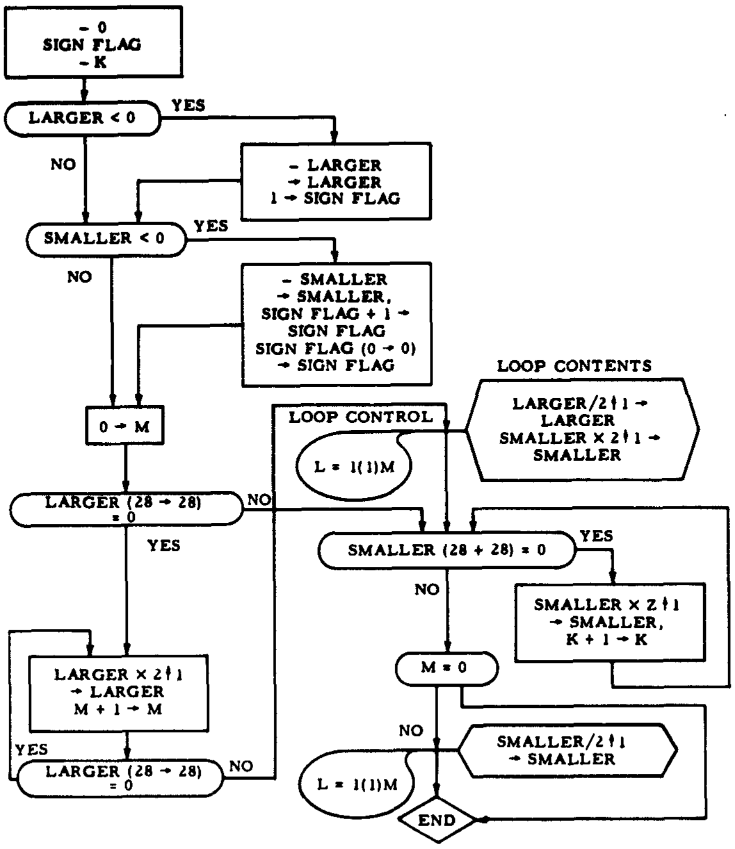


FIG. 2. Suggested Flowchart for Exercise 5.

to be the needed K in Exercise 3 and store QUOTIENT in the reserve matrix Ei, making sure that Ei has the correct sign by utilizing SIGN FLAG. If Ai < Bi, find the sin(Ci) and store it in the reserve matrix Di.

EXERCISE 7

Utilizing the sine function in Exercise 4

$$\cos Z = \sin x \text{ when } Z = \frac{\pi}{2} - x, \text{ and } \tan x = \frac{\sin x}{\cos x}$$

generate a complete trigonometric table from 0 to 0.785 radian (i.e., π/4) at intervals of 0.001 radian. Print out all parameters to five decimal places. Make the top line of the first page (in the center) "TRIGONOMETRIC TABLES," leave three blank lines and head seven columns with THETA, SIN, COS, TAN, COT, SEC, CSC. Skip one line and then list the answers so that the rightmost decimal places fall directly below the rightmost letter in the heading. List all future rows in single spacing.

Caution: COT and CSC are not defined at 0.

SOLUTIONS TO EXERCISES

```
5                    (COMMENT: EXERCISE 1)
X1, X2, X3, X4, X5, X6,
X(6).
B: {B UPPER(15→29), B LOWER(0→14)} (100)= 9,,,14,
A(100),
LOCATION = { B };..

5                    (COMMENT: EXERCISE 2)

[FMAT: 2I20,],
A(100), B(100), TEMP STORAGE,
{LOCATION OF A(0→14)} = {A}, {LOCATION OF B(0→14)} = {B};
SORT(LOCATION OF A), SORT(LOCATION OF B),
I = 0(1)99{PRINT(FMAT, A[I], B[I]),},
[STOP<,],
SORT(ADDRESS OF VECTOR):
{ADDRESS OF VECTOR → K,
I = K + 1(1)K+99{[I] → TEMP STORAGE, I - 1 → J,
TEST FOR POSITION:
TEMP STORAGE ≥ [J]: TEMP STORAGE → [J + 1];
{[J] → [J + 1], J ≠ ADDRESS OF VECTOR: J - 1 → J,
TEST FOR POSITION. TEMP STORAGE → [K];};},} . .
```

```
5                  (COMMENT: EXERCISE 3)
REMAINDER ;
DIVIDE( DIVIDEND, DIVISOR, SIGN FLAG ; QUOTIENT ):
{ 0 → QUOTIENT,
I = 0(1)K { DIVIDEND - DIVISOR → REMAINDER ≥ 0 :
REMAINDER → DIVIDEND, QUOTIENT ×2↑1 + 1 → QUOTIENT ;
QUOTIENT×2↑1 → QUOTIENT; DIVISOR /2↑1 → DIVISOR, } ,
DIVIDEND - DIVISOR > 0 : QUOTIENT + 1 → QUOTIENT ; ;
IF SIGN FLAG ≠ 0 : -QUOTIENT → QUOTIENT ; ; } ..

5                  (COMMENT: EXERCISE 4)
C|(5) = 1.5707963, -0.64596371, 0.07968968,
       -0.00467377, 0.00015148,
S|IN Z. X|. X|SQUARE. T|WO DIV BY PI = 0.63661977,
I|STORE ;
(COMMENT: TEMPORARY NAMES USED SINCE THIS IS A
          GENERAL PURPOSE FUNCTION.)

SIN(Z.):
{ i → ISTORE, Z × TWO DIV BY PI → X , X×X → XSQUARE ,
0 → SIN Z,
I = 4(-1)1{(SIN Z + C[I] ) × XSQUARE → SIN Z, } ,
ISTORE → i,
(SIN Z + C[0] ) × X → SIN Z , SIN Z → SIN Z, } , ..
```

```
5                 (COMMENT: EXERCISE 5)
;
JUSTIFY AND FIND DIV SIGN( LARGER, SMALLER ;
 SIGN FLAG, LARGE, SMALL ) :
{ 0 → SIGN FLAG → K,
LARGER < 0 : -LARGER → LARGER , 1 → SIGN FLAG ; ;
SMALLER < 0 : -SMALLER → SMALLER , SIGN FLAG + 1
 → SIGN FLAG , SIGN FLAG (0 → 0) → SIGN FLAG ; ;
0 → M , LARGER(28 → 28) ≠ 0 : JUSTIFY SMALLER. ;

JUSTIFY LARGER :
LARGER ×2↑1 → LARGER , M + 1 → M ,
LARGER(28 → 28) = 0 : JUSTIFY LARGER. ;
L = 1(1)M{LARGER/2↑1 → LARGER , SMALLER ×2↑1 → SMALLER},

JUSTIFY SMALLER :
SMALLER( 28 → 28 ) = 0 : K + 1 → K ,
SMALLER ×2↑1 → SMALLER, JUSTIFY SMALLER . ;
K ≠ 0 : L = 1(1)M { SMALLER /2↑1 → SMALLER ,},;;
SMALLER → SMALL, LARGER → LARGE  , } ..
```

```
5                   (COMMENT: EXERCISE 6)

       A(100), B(100), C(100). D(100). E(100),
       TEMP STORAGE,
       {LOCATION OF A (0 → 14) } = { A },
       {LOCATION OF B (0 → 14) } = { B },
       SIGN FLAG ;

MAIN PROGRAM :
       SORT( LOCATION OF A ),
       SORT( LOCATION OF B ),
I=0(1)99{ABSF( A[I] ) ≥ ABSF( B[I] ) :
       JUSTIFY AND FIND DIV SIGN( A[I], B[I] ;
       SIGN FLAG, A[I], B[I] ),
       DIVIDE( A[I], B[I] , SIGN FLAG ; E[I] ), ;
       SIN( C[I] ) → D[I] ;},
[STOP<,],

ABSF(ARG):
       { IF ARG < 0 : 14000₈0 ; ; } ..
```

```
5                  (COMMENT: EXERCISE 7)
COSINE. P. Q.
PI OVER 2 = 1.5707963,
[FORMAT1 :|54|<TRIGONOMETRIC TABLES>////|14|<THETA>|15|
 <SIN>|15|<COS>|15|<TAN>|15|<COT>|15|<SEC>|15|<CSC>/],
[FMAT  2: F19.5, 3F18.5, F36.5,],
[FMAT  3: F19.5, 6F18.5,],
H = 0.001,
SIN X. COS X. TAN X. COT X. SEC X. CSC X. ;

PRINTING EXERCISE:
       PRINT(FORMAT 1 ) ,
       0.0 → P → SIN X → TAN X, 1.0 → COS X → SEC X,
       PRINT(FMAT 2 , P, SIN X, COS X, TAN X, SEC X),
I=1(1)785{ P + H → P, SIN(P) → SIN X, COS(P) → COS X,
       SIN X / COS X → TAN X, 1.0 / SIN X → CSC X ,
       1.0  / COS X → SEC X , 1.0 / TAN X → COT X ,
       PRINT(FMAT 3, P, SIN X, COS X, TAN X,
       COT X, SEC X, CSC X),},
[STOP<,],

COS(A.):
       {PI OVER 2 - A → Q , SIN(Q) → COSINE ,}..
```

Appendix C
NELIAC INTERNAL CODE

Octal	Char-acter	Decimal	Octal	Char-acter	Decimal
00	Space	00	40	5	32
01	A	01	41	6	33
02	B	02	42	7	34
03	C	03	43	8	35
04	D	04	44	9	36
05	E	05	45	'	37
06	F	06	46	,	38
07	G	07	47	;	39
10	H	08	50	.	40
11	I	09	51	:	41
12	J	10	52	(	42
13	K	11	53	)	43
14	L	12	54	[	44
15	M	13	55	]	45
16	N	14	56	{	46
17	O	15	57	}	47
20	P	16	60	=	48
21	Q	17	61	≠	49
22	R	18	62	≥	50
23	S	19	63	<	51
24	T	20	64	≤	52
25	U	21	65	>	53
26	V	22	66	→	54
27	W	23	67	+	55
30	X	24	70	-	56
31	Y	25	71	/	57
32	Z	27	72	×	58
33	0	28	73	Not used	59
34	1	29	74	|	60
35	2	30	75	∪	61
36	3	30	76	∩	62
37	4	31	77	↑	63

Appendix D
REVIEW OF PUNCTUATION

A discussion per se of the six punctuation operators

$$, \; ; \; . \; : \; \} \; \{$$

has been omitted intentionally from the foregoing text since it is felt that correct usage of punctuation is largely dependent on the understanding of the CO-NO operations and not simply on the memorizing of a set of rules. However, it is often helpful to have such a set of rules available for reference.

COMMAS

The comma is often referred to as the universal separator since it is used to indicate the end of an arithmetic sequence of steps. Commas may be used with great freedom.

A comma may also indicate the call of a subroutine when the previous operator is also a punctuation symbol.

Example

; SORT SUBROUTINE,

SEMICOLONS

The semicolons perform precisely the same operations as the comma and can also indicate the end of a true or false alternative of a decision.

Example

$A < B$: SORT SUBROUTINE ; $A \rightarrow B$;

Since the semicolon does everything the comma could do, the comma is unnecessary (although not illegal) after the subroutine call or the store operation. This example could have legally been written as

$A < B$: SORT SUBROUTINE , ; $A \rightarrow B$, ;

The semicolon is also used to separate input and output parameters in the list of arguments of a subroutine.

PERIODS

The period has all the properties of a semicolon (except as an input-output separator in the list of dummy variables for a subroutine) and, in addition, will generate an unconditional transfer when the previous operator is a punctuation symbol.

COLONS

The colon has two uses. When it is preceded by a punctuation symbol, it indicates the definition of the name of a transfer point. When it is preceded by a decision operator, it signifies the beginning of the true alternative.

Examples

, THIS IS A TRANSFER POINT:	(Definition)
, $A < B : A \rightarrow B$	(Beginning of true alternative)

BRACES

Left and right braces are used as grouping symbols. They can enclose:

1. An entire true or false alternative
2. A loop
3. A subroutine or function

The braces have all the powers of a comma.

Example

$i = 0\ (1)\ 10 \quad \{A\,[i] \rightarrow B\,[i]\}$ (Right brace acts as a comma as well as ending the loop)

Appendix E

DEBUGGING AIDS AVAILABLE FROM THE NELIAC COMPILER

As NELIAC compiles a program, it lists the grammatical errors in the form of a program diagnostic on the operator's console. NELIAC will attempt to compile all of a flowchart regardless of previous errors. Although the program produced will probably be useless (if an error is diagnosed), this will minimize the number of compilation attempts needed to produce the first grammatically correct flowchart.

Errors will be categorized into one of the 14 diagnostic statements listed below. Obviously, there will be instances in which these diagnostics are slightly misleading and the programmer will need to develop a kind of sixth sense about them.

Diagnostic Statements From the Compiler

1. SHORT WORKING SPACE.
2. CAUTION, FOLLOWING NAMES UNDEFINED.
3. UNDEFINED ROUTINE, JUMPED TO FROM $xxxxxx_8$ ($xxxxxx_8$ will be the object program location).
4. STRAIGHT JUMP TO SUBROUTINE.
5. ILLEGAL CO/OPERAND/NO (followed by the flowchart in which it appears).
6. ILLEGAL DIMENSIONING STATEMENT, PROBABLY FORGOT ;
7. NAME LIST OVERFLOW (maximum of 1000_8 names for any set of flowcharts).
8. MISSING ALTERNATIVE.
9. TREATED AS FULL WORD FIXED POINT.
10. --NAME-- USED TWICE. (--NAME-- is a doubly defined noun or verb or a noun and a verb.)
11. MACHINE CODING FAULT.
12. MISSING BRACE.
13. TOO MANY UNDEFINED NAMES. (Too many names have been used previous to their definition; the maximum allowable is 128).
14. CORE EXCEEDED.

Once a flowchart has been compiled, there is always a distinct possibility that the programmer made logical errors that invalidate the results. Thecompiler has three forms of output to assist him in finding these errors:

1. Name List. The list of all names (nouns, transfer points, functions and subroutines) and their corresponding absolute addresses in memory for use with the machine code dump.

2. Machine Code Dump. Any part (or all) of memory can be dumped in octal through the compiler. The machine code generated by NELIAC can be obtained in this manner.

3. NELIAC Dump. The compiler can edit the original flow-chart and output the edited version through the high-speed punch or printer. This version will be a much more readable form than the original.

Appendix F

INITIAL NOTS CONFIGURATION OF THE U-490

In the NOTS version of the UNIVAC 490, there are 16,376 locations for storage of 30 bits each.

Peripheral equipment for the U-490 consists of four magnetic-tape transports, a high-speed printer, a typewriter, and a high-speed paper tape unit (which reads or punches a tape). In addition, there are off-line Flexowriters available for manual preparation of paper tapes.

Output from the computer is in one of the following four forms:

1. Printing from the high-speed printer
2. Magnetic tapes
3. Printing on the typewriter
4. Paper tape from the high-speed punch

In general, the first of these represents the normal mode for general-purpose, scientific programs.

There are three normal modes of input:

1. Paper tape that was manually prepared on an off-line Flexowriter or that was an output (via the high-speed punch) from a previously run program.

2. Magnetic tape that was previously prepared by a U-490 program. Note that there will be no way of preparing a magnetic tape except through the computer.

3. Manual entry through the typewriter on the console.

The normal mode of input for scientific problems is the manually prepared paper tape (Item 1, above).

INITIAL DISTRIBUTION

5 Chief, Bureau of Naval Weapons
  DLI-31 (2)
  R-14 (1)
  RU (1)
  RUTO (1)
1 Chief, Bureau of Ships (Code 560)
1 Chief of Naval Operations
2 Chief of Naval Research
  Code 104 (1)
  Code 466 (1)
1 David W. Taylor Model Basin
1 Naval Air Development Center, Johnsville
1 Naval Ordnance Laboratory, Corona
2 Naval Ordnance Laboratory, White Oak
  Library, Dr. S. J. Raff (1)
1 Naval Postgraduate School, Monterey (Library, Technical Reports Section)
2 Naval Research Laboratory
  Code 5550 (1)
1 Naval Torpedo Station, Keyport (Quality Evaluation Laboratory, Technical Library)
1 Naval Underwater Ordnance Station, Newport
1 Naval War College, Newport
2 Naval Weapons Services Office
5 Navy Electronics Laboratory, San Diego
  Code 1730 (2)
  Code 2030 (2)
1 Navy Mine Defense Laboratory, Panama City
1 Navy Underwater Sound Laboratory, Fort Trumbull
10 Armed Services Technical Information Agency (TIPCR)
1 Scientific and Technical Information Facility, Bethesda (NASA Representative S-AK/DL)
1 Applied Physics Laboratory, University of Washington, Seattle
1 Bell Telephone Laboratories, Murray Hill, N.J.
1 California Institute of Technology, Pasadena (Library)
1 Control Technology, Inc., Long Beach (A. M. Bradley)
1 Hudson Laboratories, Columbia University, Dobbs Ferry, N.Y.
1 Jet Propulsion Laboratory, CIT, Pasadena (P.R. Peabody)
1 Ordnance Research Laboratory, Pennsylvania State University (Development Contract Administrator)
1 Scripps Institution of Oceanography, University of California, La Jolla (Document Control)
1 Woods Hole Oceanographic Institution, Woods Hole, Mass.

ABSTRACT CARD

U. S. Naval Ordnance Test Station
NELIAC Programming Primer for U-490, by Dean W. Lawrence. China Lake, Calif., NOTS, May 1963. 42 pp. (NAVWEPS Report 8044, NOTS TP 3038), UNCLASSIFIED.

(Over)
1 card, 4 copies

U. S. Naval Ordnance Test Station
NELIAC Programming Primer for U-490, by Dean W. Lawrence. China Lake, Calif., NOTS, May 1963. 42 pp. (NAVWEPS Report 8044, NOTS TP 3038), UNCLASSIFIED.

(Over)
1 card, 4 copies

U. S. Naval Ordnance Test Station
NELIAC Programming Primer for U-490, by Dean W. Lawrence. China Lake, Calif., NOTS, May 1963. 42 pp. (NAVWEPS Report 8044, NOTS TP 3038), UNCLASSIFIED.

(Over)
1 card, 4 copies

U. S. Naval Ordnance Test Station
NELIAC Programming Primer for U-490, by Dean W. Lawrence. China Lake, Calif., NOTS, May 1963. 42 pp. (NAVWEPS Report 8044, NOTS TP 3038), UNCLASSIFIED.

(Over)
1 card, 4 copies

NAVWEPS Report 8044

ABSTRACT. This report serves as an introduction to computer programming in the NELIAC language. Specifically, it is oriented toward the UNIVAC-490 digital computer, a facility of the Naval Ordnance Test Station. The language is precisely designed to translate scientific problems into coding acceptable to the machine. The resulting information is useful in the design and development of antisubmarine weapons.

NAVWEPS Report 8044

ABSTRACT. This report serves as an introduction to computer programming in the NELIAC language. Specifically, it is oriented toward the UNIVAC-490 digital computer, a facility of the Naval Ordnance Test Station. The language is precisely designed to translate scientific problems into coding acceptable to the machine. The resulting information is useful in the design and development of antisubmarine weapons.

NAVWEPS Report 8044

ABSTRACT. This report serves as an introduction to computer programming in the NELIAC language. Specifically, it is oriented toward the UNIVAC-490 digital computer, a facility of the Naval Ordnance Test Station. The language is precisely designed to translate scientific problems into coding acceptable to the machine. The resulting information is useful in the design and development of antisubmarine weapons.

NAVWEPS Report 8044

ABSTRACT. This report serves as an introduction to computer programming in the NELIAC language. Specifically, it is oriented toward the UNIVAC-490 digital computer, a facility of the Naval Ordnance Test Station. The language is precisely designed to translate scientific problems into coding acceptable to the machine. The resulting information is useful in the design and development of antisubmarine weapons.