Standards

Editor's Note: The following description of NELIAC language is representative of a group of NELIAC compilers. As yet, there is not a standard NELIAC. However, at a NELIAC User's Conference in January, 1963 there was a decision to develop such a standard.—J.G.

S. GORN, Editor
R. W. BEMER, Assistant Editor, Glossary & Terminology
J. GREEN, Assistant Editor, Programming Language

A Syntactic Description of BC NELIAC

H. D. Huskey, Ralph Love and Niklaus Wirth
University of California, Berkeley

In 1958, at the time of the formation of an International Algorithmic Language for Computing Machines (subsequently named Algol), a project was started at the U. S. Naval Electronics Laboratory in San Diego to develop a translator for IAL. Overtaking the definition effort, they defined their own language, doing so with a particular control application in mind. Thus, a problem-oriented language based on Algol was defined and translators were built for a number of computers (Sperry-Rand, Datatron, CDC and IBM, among others). The resulting language (named Neliac) was algebraic in character (like Algol) but much simpler and straightforward (and consequently, much easier to learn and to use). Minimum effort principles were used in the design—thus, things that are said frequently can be said simply, and historical mathematical notations are respected as far as feasible. Using load-and-go techniques, single-pass fast compilation was possible (more than 5000 object commands per minute), and fast-running programs were obtained.

In the four years since the start of the activity, Neliac has evolved through several generations with improvement of power, speed of compiling and speed of object programs. Some versions permit nested parenthetical expressions in statements, some generate relocatable binary codes and some have elaborate input and output format and control capabilities.

All versions of Neliac are self-compiling; that is, their translators are written in Neliac. Due to fast compilation, changes to any Neliac program are always made in source language. This gives the nontrivial advantage that documentation is always up to date. Also, with 15 significant characters per identifier, a Neliac program is an easily readable document.

The version described here (called BC Neliac) is a descendant of the IBM 704-IBM 709-IBM 7090 Neliacs developed at Fort Huachucha, which themselves came from the Neliac for the Sperry-Rand M460 in San Diego. Many people were involved in this development. To mention a few: M. Halstead at the U.S. Naval Electronics Laboratory, R. Landon of Ramo-Wooldridge at Fort Huachucha, and W. Wattenburg at the University of California.

This paper defines the reference language and hardware representation for BC Neliac. The authors of this paper have made certain additions to the original IBM 704 Neliac which make it a more powerful, flexible and a more machine-independent language. Some of the features added are the following: subscripting with identifiers, logical AND and OR expressions, Algol word delimiters, character manipulating operations and absolute addressing. Both the syntax and semantics of this language are discussed and examples are given. Rules for transliteration from the reference language to the hardware representation, and a syntactical flowchart are included in the appendix.

Metalanguage for Syntactic Description

The Syntax of BC Neliac is described using the Algol metalanguage. The basic symbols of this language are:

::= metalinguistic connective meaning *is defined to be*
| metalinguistic connective meaning *or*
⟨ ⟩ delimiting brackets which enclose metalinguistic variables.

Metalinguistic variables are a sequence of characters enclosed in the delimiting brackets ⟨ ⟩. The symbols used for distinguishing the metalinguistic variables have been chosen to be words describing approximately the nature of the corresponding variable. This is done only for understanding and has no technical significance. A mark in a formula, which is not a variable, connective or delimiter, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the marks and/or variables in the language being defined. Metalinguistic formulae are composed of metalinguistic connectives, variables enclosed within delimiting brackets and an indication of juxtaposition. *Examples:*

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩
⟨letter⟩ ::= A | B | C | . . . | Y | Z
⟨digit⟩ ::= 0 | 1 | 2 | . . . | 8 | 9

The formula for identifier is recursive since ⟨identifier⟩ appears on both sides of the "defining connective." The metalinguistic variable ⟨letter⟩ indicates ⟨identifier⟩ can have the value A, or B, or C, etc. The marks ⟨identifier⟩ ⟨digit⟩ mean, given some value of ⟨identifier⟩, another can be formed by juxtapositioning a value of the variable ⟨digit⟩. If the values of digit are the arabic numerals, then the following are illustrations of legitimate values of ⟨identifier⟩:

A
AB
A1B
Y55A
XYZ799

1.0. Basic Symbols, Identifiers, and Numbers

1.0.1. *Semantics:* The language of NELIAC is constructed of the basic symbols presented in this section.

1.0.2. *Syntax:*

⟨basic symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨delimiter⟩

1.1. LETTERS

1.1.1. *Semantics:* The Roman alphabet is used for letters. The letters are used to form identifiers and do not have individual meanings. However, the letters I through N have significance with respect to subscripts (see 2.1).

1.1.2. *Syntax:*

⟨letter⟩ ::= ⟨sub letter⟩ | I | J | K | L | M | N
⟨sub letter⟩ := A | B | C | D | E | F | G | H | O | P | Q | R | S | T | U | V | W | X | Y | Z

1.2. DIGITS.

1.2.1. *Semantics:* Digits are used to form numbers and identifiers.

1.2.2. *Syntax:*

⟨digit⟩ ::= ⟨octal digit⟩ | 8 | 9
⟨octal digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

1.3. DELIMITERS.

1.3.1. *Semantics:* Most delimiters are operators or separators indicating relationships among identifiers. Blank space or change to new line have no significance in the reference language.

1.3.2. *Syntax:*

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨punctuation⟩ | ⟨specifier⟩
⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator⟩ | ⟨sequential operator⟩ | ⟨psuedo operator⟩
⟨arithmetic operator⟩ ::= ⟨binary operators⟩
⟨binary operator⟩ ::= + | − | * | / | → | ←
⟨relational operators⟩ ::= < | ≦ | = | ≠ | ≧ | >
⟨logical operators⟩ ::= ∧ | ∨
⟨sequential operators⟩ ::= GO TO | IF | FOR | DO | THEN | ELSE
⟨pseudo operators⟩ ::= OCT | EXP | MCH
⟨separator⟩ ::= . | , | : | ; | STEP | UNTIL
⟨bracket⟩ ::= (|) | [|] | BEGIN | END
⟨punctuation⟩ ::= , | . | ;
⟨specifier⟩ ::= REAL | INTEGER | PROCEDURE | ⟨specifier⟩ ARRAY | ⟨empty⟩

1.4. IDENTIFIER.

1.4.1. *Semantics:* An identifier is a string of letters and digits that begins with a letter. Blank spaces may appear as part of an identifier. No delimiters may be used with an identifier. Identifiers may designate variables or may be used as labels in a program. If an identifier is used as a label it may be considered to apply to either the point in the program at which it occurs, or it may be thought of as the name of all the section of coding which follows.

Restrictions: Only the first 15 characters, not counting spaces, are significant in identifiers.

1.4.2. *Examples:*

Q
D06
G15
IBM 704
FOUR
TRUE

1.4.3. *Syntax:*

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩
⟨procedure identifier⟩ ::= ⟨declaration identifier⟩
⟨label⟩ ::= ⟨declaration identifier⟩
⟨declaration identifier⟩ ::= ⟨sub letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩

1.5. NUMBERS.

1.5.1. *Semantics:* Three types of numbers are allowed: fixed point or integer, floating point or real, and octal. Numbers which are real must include a decimal point and power of 10.

Restriction: Only ten digits are significant.

1.5.2. *Examples:*

0
0.0*0
−26
94.3* − 2
0.943* − 2
OCT 74476
+5

1.5.3. *Syntax:*

⟨number⟩ ::= ⟨unsigned number⟩ |
+ ⟨unsigned number⟩ |
− ⟨unsigned number⟩
⟨unsigned number⟩ ::= ⟨unsigned integer⟩ |
⟨real number⟩
⟨real number⟩ ::= ⟨decimal integer⟩
⟨decimal fraction⟩⟨exponent part⟩
⟨unsigned integer⟩ ::= ⟨decimal integer⟩ |
OCT⟨octal integer⟩
⟨decimal integer⟩ ::= ⟨digit⟩ |
⟨decimal integer⟩⟨digit⟩
⟨octal integer⟩ ::= ⟨octal digit⟩ |
⟨octal integer⟩⟨octal digit⟩
⟨decimal fraction⟩ ::= .⟨decimal integer⟩
⟨exponent part⟩ ::= *⟨signed integer⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ |
⟨signed integer⟩
⟨signed integer⟩ ::= + ⟨unsigned integer⟩ |
− ⟨unsigned integer⟩

1.6. COMMENTS.

1.6.1. *Semantics:* Comments may be inserted at any arbitrary place in the program. They have no influence on the meaning of the program.

1.6.2. *Example:*

(COMMENT—NELLAC SENTENCE FORMAT)

1.6.3. *Syntax:*

⟨comment⟩ ::= (COMMENT ⟨any sequence of letters, digits and delimiters except a right parenthesis⟩)

2.0. Variables

2.1. *Semantics:* A variable is used in BC NELIAC to denote any quantity that is referred to by name. The variable may be used in expressions and its value may be changed by means of an assignment statement. Variables can be declared as fixed-point or floating-point quantities.

A subscripted variable designates values which are components of linear or single dimensional arrays. The array component referred to by a subscripted variable is specified by the actual numerical value of the subscript expression and is an integer.

Variables may specify subfields of a complete computer word by specification of the low and high order bit numbers. Also, variables may be considered as consisting of a chain of characters or symbols in which each character is treated as a separate entity.

The letters I through N are reserved for variables of a particular type known as indices, and they must not be declared.

2.2. *Examples:*

Variables	*Subscripts*
A	[J-1]
WORD [TA6-20]	[LOCATION-5]
ADDRESS [J-10](10→15)	[N]
SYMBOL 1 [K-11](*6)	[M+5]
N	[3626]
RACK [OCT 26762]	

2.3. *Syntax:*

⟨variable⟩ ::= ⟨identifier⟩⟨subscript⟩
⟨structure specification⟩
⟨subscript⟩ ::= ⟨empty⟩ | [⟨subscript expression⟩]
⟨subscript expression⟩ ::= ⟨index⟩ |
⟨index⟩⟨signed integer⟩ | ⟨unsigned integer⟩ |
⟨signed integer⟩ | ⟨identifier⟩
⟨index⟩ ::= I | J | K | L | M | N
⟨structure specification⟩ ::= ⟨part word limit⟩ |
⟨chain declaration⟩ | ⟨empty⟩
⟨part word limit⟩ ::= (⟨unsigned integer⟩ →
⟨unsigned integer⟩)
⟨chain declaration⟩ ::= (*⟨unsigned integer⟩)

3.0. Expressions

3.1. ARITHMETIC EXPRESSIONS

3.1.1. *Semantics:* Arithmetic expressions are used to compute a numerical value by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expressions. The arithmetic expression is followed by a left to right arrow to denote replacement and a variable which is set equal to the value of whatever preceded the arrow. Part word, subscripted variables and subscripted part words can also be used as variables in the expression. A function, which has a number of inputs and one output, can be called any place within an arithmetic expression. In binary computers the pseudo operator EXP, used in arithmetic expressions in the form 2 EXP *n*, will shift the contents of an arithmetic expression *n* bits to the right or left depending on the multiplying operator preceding the 2. The operator / shifts the contents to the right while the operator * shifts the contents to the left. The normal order of processing is: exponentiation is done first, followed by multiplication and division, followed finally by addition and subtraction. Where otherwise ambiguous, processing is from left to right.

Restriction: In BC NELIAC real and integer quantities cannot be mixed and parenthetical grouping is not allowed in arithmetic expressions.

3.1.2. *Examples:*

A/5
A + B
A * 2 EXP 5
B + 24 * D(R, S, T+V) + I

3.1.3. *Syntax:*

```
⟨arithmetic expression⟩ ::= ⟨term⟩ |
    ⟨arithmetic expression⟩⟨adding operator⟩
    ⟨term⟩
  ⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩
      ⟨multiplying operator⟩⟨factor⟩
  ⟨adding operator⟩ ::= + | −
    ⟨factor⟩ ::= ⟨primary⟩ | ⟨primary⟩ * 2 EXP
        ⟨unsigned integer⟩ | ⟨primary⟩/2 EXP
        ⟨unsigned integer⟩
    ⟨multiplying operator⟩ ::= * | /
      ⟨primary⟩ ::= ⟨unsigned number⟩ |
          ⟨variable⟩ | ⟨function designator⟩
        ⟨function designator⟩ ::= ⟨procedure
            identifier⟩⟨actual parameter part⟩
          ⟨actual parameter part⟩ ::= (⟨actual
              parameter list⟩)
            ⟨actual parameter list⟩ ::=
            ⟨parameter⟩ | ⟨actual parameter
                list⟩, ⟨parameter⟩
              ⟨parameter⟩ ::= ⟨expression⟩
                ⟨expression⟩ ::= ⟨arithmetic
                    expression⟩ |
                    ⟨chain expression⟩ |
                    ⟨logical expression⟩
```

3.2. CHAIN EXPRESSIONS

3.2.1. *Semantics:* Chain expressions are intended to simplify operations involving character manipulations. A variable will consist of a chain of characters or symbols when a chain declaration is applied to it. Two operations may be performed on chain variables: "catenate" and "obtain first character".

The catenating operation will adjoin one character at the right of a chain variable.

The obtain first character operation will get the left-most character of a chain variable and has the form "* variable followed by a binary operator."

3.2.2. *Examples:*

```
A + + B
*A + + D
A/D − B + + R
```

3.2.3. *Syntax:*

```
⟨chain expression⟩ ::= ⟨expression⟩
  ++ ⟨variable⟩ | * ⟨expression⟩
```

3.3. LOGICAL EXPRESSION (available in binary computer)

3.3.1. The logical operation of two variables is performed in the logical expression. Logical operations are frequently used in a process called "masking". This is the process of extracting one or more small parts of a word from the whole word.

The AND and OR concept is used with logical operations. When two primaries are combined by an AND, they are matched bit-for-bit according to the following truth table.

Bit of primary 1	*Bit of primary 2*	*Resulting* AND *bit*
0	0	0
0	1	0
1	0	0
1	1	1

An OR function also matches two primaries bit-for-bit. The following truth table applies.

Bit of primary 1	*Bit of primary 2*	*Resulting* OR *bit*
0	0	0
0	1	1
1	0	1
1	1	1

3.3.2. *Examples:*

```
TAG ∧ ADDRESS ∨ SYMBOL
OPCODE ∧ 64 ∧ STORAGE
```

3.3.3. *Syntax:*

```
⟨logical expression⟩ ::= ⟨primary⟩ |
    ⟨logical expression⟩⟨logical operator⟩⟨primary⟩
  ⟨logical operator⟩ ::= ∧ | ∨
```

3.4. RELATION

3.4.1. *Semantics:* A relation is a logical or Boolean variable used in a conditional statement. If the condition stated is satisfied, the value of the relation is true; otherwise, it is false.

3.4.2. *Examples:*

```
A ≠ B
A < B < C
A + C < D < E
2 ≦ X ≦ Y ≦ 10
```

3.4.3. *Restriction:* In a relation it is necessary that all relational operators be the same.

3.4.4. *Syntax:*

```
⟨relation⟩ ::= ⟨simple relation⟩ | ⟨relation⟩
    ⟨relational operator⟩⟨variable⟩
  ⟨simple relation⟩ ::= ⟨expression⟩
      ⟨relational operator⟩⟨variable⟩
    ⟨expression⟩ ::= ⟨arithmetic expression⟩ |
        ⟨chain expression⟩ | ⟨logical expression⟩
```

4.0. Statements

4.0.1. *Semantics:* Basic statements are the units of instruction in the NELIAC language. According to different tasks, there exist different types of basic statements, all of which may additionally be labeled. They are described in the following Sections 4.1–4.6.

Basic statements may be catenated. Normally a comma is written as a punctuation symbol in between basic statements; however, for clarity a period may be placed after a GO TO statement. A sequence of basic statements separated by commas is called an unconditional statement.

Unconditional statements may be prefixed by conditions, thus forming a conditional statement (4.7); furthermore, a sequence of unconditional or conditional statements may be enclosed between the words BEGIN and END, Thus forming a compound statement (4.8). Since a compound statement is considered as an unconditional statement, it presents the means for cascading conditions. After the delimiter PROCEDURE a labeled compound statement is considered as a procedure body.

4.0.2. *Syntax:*

```
⟨statement⟩ ::= ⟨unconditional statement⟩, |
     ⟨conditional statement⟩ |
     ⟨label⟩: ⟨statement⟩ | ⟨procedure body⟩,
  ⟨unconditional statement⟩ ::=
       ⟨basic statement⟩ | ⟨basic statement⟩,
       ⟨unconditional statement⟩ |
       ⟨compound statement⟩
     ⟨basic statement⟩ ::= ⟨assignment statement⟩|
       ⟨go to statement⟩ | ⟨procedure statement⟩ |
       ⟨for statement⟩ | ⟨dummy statement⟩ |
       ⟨code⟩ | ⟨label⟩:⟨basic statement⟩
```

4.1. Assignment Statements

4.1.1. *Semantics:* The assignment statement specifies an expression which is to be evaluated and a variable which is to have the resulting value assigned to it. All variables must be declared to be of the same type. If the variable to the right of an arrow is designating a partial word, then the part(s) of the word not designated remain unaffected by the assignment statement.

An assignment statement is executed in the following steps: (1) The expression to the left of the arrow is evaluated. (2) The subscript expression of the variable to the right of the left-most arrow is evaluated. (3) The variable is assigned the value of the expression. (4) For each following variable steps 2 and 3 are performed sequentially.

4.1.2. *Examples:*

```
A → B
X[I] + X[J] → A → B
2 → B
```

4.1.3. *Syntax:*

```
⟨assignment statement⟩ ::= ⟨expression⟩⟨right
    part list⟩
  ⟨expression⟩ ::= ⟨arithmetic expression⟩ |
      ⟨chain expression⟩ | ⟨logical expression⟩
    ⟨right part list⟩ ::= ⟨right part⟩ | ⟨right part
        list⟩⟨right part⟩
      ⟨right part⟩ ::= → ⟨variable⟩
```

4.2. GO TO Statements

4.2.1. *Semantics:* Unconditional transfer of control statements are formed by following the words GO TO with an identifier. Thus the next statement to be executed will be the one being labeled with this identifier. A switch statement may be referenced by a GO TO statement which is subscripted. A switch statement consists of a label by which it is referenced, and a list of alternative points in a program to which control may be transferred. The selection of the actual point to which control is transferred depends on the value of the subscript expression in the GO TO statement.

4.2.2. *Restriction:* A GO TO statement may not refer to a label occurring within a compound statement unless the GO TO statement is part of the compound statement itself. Neither may a GO TO statement refer to the label of that compound statement (procedure body) inside which it stands.

4.2.3. *Examples:*

```
GO TO EUROPE
GO TO WINDOW [15]
```

4.2.4. *Syntax:*

```
⟨go to statement⟩ ::= GO TO ⟨label⟩⟨subscript⟩
```

4.3. Compound Statements

4.3.1. *Semantics:* The compound statement consists of a group of statements which are separated by punctuation marks and enclosed by the words BEGIN and END. It is generally of the form

BEGIN S, S, ... S END,

where S represents arbitrary statements, which themselves may be compound statements. In order to facilitate reading of a program containing nested compound statements, the label of a compound statement (if there is one) may be repeated after the closing word END. This identifier will then be regarded as if it were a comment. After the word delimiter PROCEDURE a labeled compound statement is called a procedure body. A procedure may not be called recursively (see Section 4.2.2.).

4.3.2. *Examples:*

```
BEGIN A + B → X, C + D → Y END
BEGIN IF A ≦ B THEN
  BEGIN IF A = B THEN GO TO NEW YORK.
  ELSE WEST; END; ELSE EAST; END
PROCEDURE P: BEGIN 0 → X → Y END P
```

4.3.3. *Syntax:*

```
⟨compound statement⟩ ::= BEGIN ⟨compound
    tail⟩ END
⟨procedure body⟩ ::= PROCEDURE ⟨procedure
    identifier⟩:⟨compound statement⟩
  ⟨compound tail⟩ ::= ⟨statement⟩ | ⟨statement⟩
    ⟨compound tail⟩
```

4.4. Procedure Statements

4.4.1. *Semantics:* A procedure is a part of a program that is written only once but is to be executed at several points throughout the same program. A procedure is called by a procedure statement which effectively inserts the procedure body into the program taking the place of the procedure statement. If the actual parameter part of the procedure statement is not empty, then the formal parameters of the called procedure body must be previously declared. This must be done in a declaration list immediately preceding the procedure body, which therefore takes the form of a flowchart. The variables in the declaration list of this flowchart are assigned the values of the corresponding actual parameters of the procedure statement. The correspondence is established by taking the entries of the actual parameter list in the same order as the variables on the declaration list. This implies that the number of actual parameters is less than or equal to the number of variables declared.

4.4.2. *Examples:*

```
GET SYMBOL
FIND RESIDENT (CITY, STREET, NUMBER)
```

4.4.3. *Syntax:*

```
⟨procedure statement⟩ ::=
  ⟨procedure identifier⟩ ⟨actual parameter part⟩ |
  ⟨procedure identifier⟩
```

4.5. Dummy Statement

4.5.1. *Semantics:* A dummy statement may serve to place a label or a clarifying punctuation. For example, it may be used to label the end of a compound statement. The dummy statement may also occur as an empty false part in a conditional statement.

4.5.2. *Syntax:*

```
⟨dummy statement⟩ ::= ⟨empty⟩
```

4.6. Code

A basic statement may have the form of a code. Thus operations may be stated which are not otherwise describable in this language. Codes will, e.g., be used to specify input-output operations.

An instruction which represents exactly one machine instruction is written in the following form:

```
MCH XXXXXXX ⟨address⟩
⟨address⟩ ::= ⟨unsigned integer⟩ | ⟨identifier⟩
```

The seven octal digits XXXXXXX represent the prefix, decrement and tag parts of the IBM 704/7090 instruction in octal form.

Example:

```
MCH 4534004 INDEX   means   LXD INDEX, 4
```

4.7. Conditional Statement

4.7.1. *Semantics:* Conditional statements cause statements to be executed or skipped depending on the current values of the specified relation. The conditional statement consists of a relation preceded by the word IF and followed by the word THEN, a "true part", and a "false part". Both "true" and "false parts" are unconditional statements. They are normally terminated by a semicolon.

If the relation is true, the statement following THEN is executed, after which control is transferred to the beginning of the next statement following the "false part", unless the "true part" terminates with a GO TO statement. If the relation is false, the "false part" is executed, after which control is transferred to the beginning of the next statement unless a GO TO statement terminates the "false part". Either "true" or "false parts" may be left vacuous by immediately terminating it with a semicolon (dummy statement).

4.7.2. *Examples:*

```
IF X + Y = Z THEN GO TO A;
  ELSE X − Y → Z;
IF B1 ∧ B2 ∨ B3 = TRUE THEN SET FLAGS;
IF LL < X < UL THEN TRUE → FLAG;
  ELSE FALSE → FLAG, GO TO ERROR;
IF 2 ≦ X ≦ Y ≦ 8 THEN BEGIN IF X = Y
  THEN GO TO S1; ELSE;; ELSE GO TO S2;
```

4.7.3. *Syntax:*

```
⟨conditional statement⟩ ::= IF ⟨relation⟩ THEN
  ⟨unconditional statement⟩; ELSE ⟨unconditional
  statement⟩;
  |IF ⟨relation⟩ THEN ⟨unconditional statement⟩;
```

4.8. FOR Statements

4.8.1. *Semantics:* The FOR statement facilitates writing an iterative operation. The index specified in the FOR clause takes on values beginning with a first limit and is modified by an increment for each successive execution of the iterative operation. The execution of the FOR statement ends when a successive application of the increment would cause the index to pass beyond the second limit. The limiting values cannot be varied from within the FOR statement.

4.8.2. *Examples:*

FOR K = INITIAL VALUE STEP 3 UNTIL FINAL VALUE DO BEGIN FIRST FLAG [K] ∧ SECOND FLAG [K] → FLAG [K] END

FOR N = 0 STEP 1 UNTIL 100 DO BEGIN VECTOR 1 [N] * VECTOR 2 (N) + SUM → SUM END

4.8.3. *Syntax:*

⟨for statement⟩ ::= ⟨for clause⟩⟨compound statement⟩
⟨for clause⟩ ::= FOR ⟨index⟩ = ⟨for limit⟩ STEP ⟨integer⟩ UNTIL ⟨for limit⟩ DO
⟨for limit⟩ ::= ⟨identifier⟩ | ⟨integer⟩ | ⟨identifier⟩⟨signed integer⟩
⟨compound statement⟩ ::= BEGIN ⟨compound tail⟩ END
⟨compound tail⟩ ::= ⟨statement⟩ | ⟨statement⟩⟨compound tail⟩

5.0. Declaration Lists

5.0.1. *Semantics:* Declarations occur at the beginning of a flowchart and serve to define certain properties of the variables in the program. The declarations will be valid for the entire program, not only for the particular flowchart in which they are declared. All identifiers used in the program except labels and indices must be declared.

The declaration list consists of specifiers to define the types of variables named by identifiers and a declaration to define certain properties of the identifiers. The declaration may consist of a declaration identifier, alternate name, structure declaration and value list.

The declaration identifier is the name by which the declared variable will be referred. If more than one name is given to the identical variable, alternate names may be listed with a colon in between. The structure declaration contains information about the substructure of the variable, which may consist of a chain of characters or of several part words. The structure declaration may be combined with the alternate name feature, such that different names refer to the same word but assign different structures to it. By means of the value list a numerical value can be preassigned to the variable before execution of the program.

5.0.2. *Examples:*

INTEGER A, B, C, REAL X, Y, Z

INTEGER A, REAL ARRAY R (3) ← 1.1 * 2, 2.5 * 7, −0.33 * 3

5.0.3. *Syntax:*

⟨declaration list⟩ ::= ⟨specifier⟩⟨declaration⟩ | ⟨declaration list⟩, ⟨declaration list⟩ | ⟨empty⟩
⟨specifier⟩ ::= REAL | INTEGER | ⟨specifier⟩ ARRAY | ⟨empty⟩ | PROCEDURE
⟨declaration⟩ ::= ⟨declaration identifier⟩⟨structure declaration⟩⟨alternate name⟩⟨value list⟩
⟨alternate name⟩ ::= ⟨empty⟩ | :⟨declaration identifier⟩⟨chain declaration⟩⟨alternate name⟩

See Section 1.4.3 for definition of declaration identifier.

5.1. STRUCTURE DECLARATIONS

5.1.1. *Semantics:* The standard field for a variable is a computer word; however, variables may represent subfields or parts of words. All names referring to part-words are included within a BEGIN and END in the declaration. Each part word name is followed by a definition of the part or subfield of the computer word and is dependent on the particular computer used. This definition is enclosed in parentheses. For the IBM 704 and IBM 7090 machines, a variable is represented by 36 bits. The part word limits then specify the right-most (lowest) and the left-most (highest) bit belonging to the named part word.

A variable may also consist of a chain of characters, symbols, or groups of bits which in the program will be treated as separate entities. In this case the number of bits forming a character or symbol, preceded by an asterisk and enclosed in parentheses, follows the declaration identifier as a chain declaration. For the IBM 704 version n has to satisfy $1 \leqq n \leqq 12$.

5.1.2. *Examples:*

LOCATION:BEGIN PREFIX (33→35), DECREMENT (18→32), TAG (15→17), ADDRESS (0→14) END

STRING (*6), CONVERTED STRING (*7)

5.1.3. *Syntax:*

⟨structure declaration⟩ ::= ⟨part word declaration⟩ | ⟨chain declaration⟩ | ⟨empty⟩
⟨part word declaration⟩ ::= : BEGIN ⟨part word list⟩ END
⟨part word list⟩ ::= ⟨declaration identifier⟩ ⟨part word limit⟩ | ⟨part word list⟩, ⟨part word list⟩
⟨part word limit⟩ ::= (⟨unsigned integer⟩→ ⟨unsigned integer⟩)
⟨chain declaration⟩ ::= (*⟨unsigned integer⟩) | ⟨empty⟩

5.2. VALUE LISTS

5.2.1. *Semantics:* The value list may preassign a numerical value to a variable and/or define the dimension of a variable in the case of an array. The value list consists of two parts, both of which may be empty. The first part defines the dimension of the variable in the case of an array (if it is empty, the dimension is assumed to be 1). The second part is the number list in the case of an array, which reduces to a number in the case of a single variable. If the number list is empty, the variable is preassigned the value 0.

As a special feature, a variable may be assigned a predetermined location in the machine (absolute addressing), by following the variable with ∗ OCT and the octal integer.

5.2.2. *Syntax:*

```
⟨value list⟩ ::= ⟨empty⟩ | ← ⟨value⟩ |
    ⟨dimension⟩ | ⟨dimension⟩ ← ⟨number list⟩ |
    ⟨absolute address⟩
  ⟨dimension⟩ ::= (⟨unsigned integer⟩)
  ⟨number list⟩ ::= ⟨value⟩ | ⟨value⟩,
      ⟨number list⟩
    ⟨value⟩ ::= ⟨integer⟩ | ⟨number⟩
  ⟨absolute address⟩ ::= ∗ OCT ⟨octal integer⟩
```

6.0. Flowcharts

6.0.1. *Semantics:* The logical segment of the NELIAC program is the flowchart. It consists of a declaration list and a compound tail. The latter is the actual program which specifies the operations to be performed on the variables defined in the declaration list.

The compound tail consists of a sequence of statements which are separated by punctuation marks (usually commas). By labeling single statements with an identifier and a colon, they may be referred to from other points of the program.[1]

Normally, statements will be executed consecutively. This rule may be broken by introducing GO TO statements which explicitly specify the next statement to be executed. Another exception to this rule is made with labeled compound statements, which are interpreted as procedure bodies and therefore are not considered a portion of the normal sequence of the program. These may be activated from any place in the program by procedure statements. The processing sequence of the program may be controlled by conditional statements, which may cause certain statements to be skipped.

A NELIAC program consists of a sequence of flowcharts. The flowcharts are not independent. Variables declared or labels occurring in any flowchart may be referred to from within any arbitrary flowchart;[1] however, normally variables should be declared before they are called.

6.0.2. *Examples:*

(a)

```
TEMP,
NUMBER (100);
SORT:
(COMMENT  A ROUTINE THAT SORTS A LIST OF 100
  UNSIGNED INTEGERS INTO NUMERIC ORDER, USING
  THE SHUTTLE EXCHANGE METHOD.)
FOR I = 0 STEP 1 UNTIL 98 DO
  BEGIN IF NUMBER [I]  >NUMBER [I+1]
    THEN BEGIN NUMBER [I] → TEMP,
      NUMBER [I+1] → NUMBER [I],
      TEMP → NUMBER [I+1], I → J,
      XX:IF J > 0
        THEN BEGIN IF NUMBER [J] < NUMBER [J−1]
          THEN NUMBER [J] → TEMP,
            NUMBER [J−1] → NUMBER [J]
            TEMP → NUMBER [J−1], J−1 → J,
            GO TO XX.
          ELSE;  END;
        ELSE;  END;
    ELSE;  END..
```

(b)

```
ARRAY X (∗6)(6), XX (6), Y (∗6)(5);
  (COMMENT   MANIPULATE  WILL  CONVERT  THE
    ARRAY X:
       A0, B0, C0, D0, E0, F0
       A1, B1, C1, D1, E1, F1
       A2, B2, C2, D2, E2, F2
       A3, B3, C3, D3, E3, F3
       A4, B4, C4, D4, E4, F4
       A5, B5, C5, D5, E5, F5  INTO THE ARRAY Y:
       B0, B1, B2, B3, B4, B5
       C0, C1, C2, C3, C4, C5
       D0, D1, D2, D3, D4, D5
       E0, E1, E2, E3, E4, E5
       F0, F1, F2, F3, F4, F5  END COMMENT)
MANIPULATE:
  MOVE, FOR J = 0 STEP 1 UNTIL 4 DO
  BEGIN MOVE, JOIN, END,
PROCEDURE MOVE:
  BEGIN FOR I = 0 STEP 1 UNTIL 5 DO
  BEGIN ∗ X [I] → XX [I],
          X [I] ++ 0 → X [I], END, END MOVE,
PROCEDURE JOIN:
  BEGIN FOR I = 0 STEP 1 UNTIL 5 DO
  BEGIN Y [J] ++ X [I] → Y [J], END, END JOIN ...
```

(c)

```
  TRA = 1, CLA = 2, STO = 3
  MEMORY (1000),
  INSTRUCTION COUNTER:IC,
  INSTRUCTION REGISTER:IR:
    BEGIN OP CODE (18→32), ADDRESS (0→14) END,
  ACCUMULATOR:AC;
(COMMENT  THIS PROGRAM SIMULATES A COMPUTER
  WITH INDIRECT ADDRESSING FACILITY AND THE
  FOLLOWING OPERATIONS: TRA, CLA, STO)
CYCLE:
  MEMORY [IC] → IR, IC + 1 → IC,
INDIRECT ADDRESS TEST:
  IF MEMORY (35→35) [ADDRESS] = 1 THEN
  MEMORY (0→14) [IR] → ADDRESS,
  GO TO INDIRECT ADDRESS TEST.  ELSE;
EXECUTE:
  IF OP CODE = TRA THEN ADDRESS → IC, GO TO CYCLE.
    ELSE;
```

---

[1] See Section 4.2.2. for restriction when referencing other points of a program.


NELIAC
SYNTACTICAL CHART
U. C. BERKELEY
Flowchart
Declaration List
Compound Tail
Statement
B 15 Compound Tail
B 12 Statement
D 24 Empty
B 17 Declaration List
B 17 Declaration List
Specifier
Declaration
Undefined
Code
Procedure Body
D 13 Compound Statement
K 6 Procedure Identifier
PROCEDURE
Label
B 12 Statement
K 6 Declaration Identifier
Unconditional Statement
C 12 Unconditional Statement
Basic Statement
Conditional Statement
IF
Relation
THEN
C 12 Unconditional Statement
ELSE
C 12 Unconditional Statement
Compound Statement
BEGIN
B 15 Compound Tail
END
D 12 Basic Statement
C 9 Label
Dummy Statement
For Statement
Procedure Statement
Go to Statement
Assignment Statement
D 24 Empty
For Clause
D 13 Compound Statement
K 6 Procedure Identifier
K 6 Procedure Identifier
K 7 Actual Parameter part
GO TO
C 9 Label
G 16 Subscript
Expression
Right part List
FOR
H 11 Index
For Limit
STEP
E 28 Integer
UNTIL
F 3 For Limit
DO
E 28 Integer
H 17 Identifier
F 27 Signed Integer
H 17 Identifier
Arithmetic Expression
F 9 Arithmetic Expression
Adding Operator
Term
G 9 Term
Multiplying Operator
Factor
H 6 Primary
Primary
G 8 Multiplying Operator
2
EXP
F 23 Unsigned Integer
F 31 Unsigned Number
Function Designator
F 16 Variable
Procedure Identifier
Actual Parameter Part
Declaration Identifier
Parameter List
H 17 Identifier
J 17 Letter or Digit
K 8 Parameter List
E 13 Expression
A
H
O
Z
Sub Letter
Chain Expression
E 13 Expression
F 16 Variable
E 13 Expression
Logical Expression
F 12 Logical Expression
Logical Operator
H 6 Primary
E 14 Right Part List
Right Part
Variable
D 24 Empty
H 17 Identifier
Subscript
Structure Specification
D 24 Empty
Subscript Expression
E 24 Part word Limit
I 21 Chain Declaration
Index
H 11 Index
F 27 Signed Integer
E 28 Integer
H 17 Identifier
Identifier
I
J
K
L
M
N
H 17 Identifier
Letter or Digit
K 17 Letter
H 23 Digit
Letter
A
B
C
Y
Z
Simple Relation
D 16 Relation
E 18 Relational Operator
E 20 Variable or Number
E 13 Expression
Relational Operator
Variable or Number
F 16 Variable
F 31 Unsigned Number
PROCEDURE
B 22 Specifier
ARRAY
REAL
Integer
K 6 Declaration Identifier
Structure Declaration
Alternate Name
Value List
Part Word Declaration
D 27 Chain Declaration
Empty
K 6 Declaration Identifier
Chain Declaration
C 27 Alternate Name
BEGIN
Part Word List
END
E 23 Part Word List
K 6 Declaration Identifier
Part Word Limit
F 23 Unsigned Integer
F 23 Unsigned Integer
Unsigned Integer
OCT
Octal Integer
H 29 Decimal Integer
G 22 Octal Integer
Octal Digit
0
1
2
3
4
5
6
7
D 34 Dimension
Number List
E 31 Value
Dimension
Absolute Address
D 24 Empty
D 31 Number List
Value
F 23 Unsigned Integer
Integer
Number
G 22 Octal Integer
OCT
Signed Integer
G 8 Adding Operator
F 23 Unsigned Integer
Unsigned Number
Real Number
F 23 Unsigned Integer
H 29 Decimal Integer
Decimal Fraction
Exponent Part
Decimal Integer
H 29 Decimal Integer
H 29 Decimal Integer
Digit
G 23 Octal Digit
8
9
A
B
C
D
E
F
G
H
J
K
L
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

```
IF OP CODE = CLA THEN MEMORY [IR] → AC, GO TO
  CYCLE. ELSE;
IF OP CODE = STO THEN AC → MEMORY [IR], GO TO
  CYCLE. ELSE . . .
```

6.0.3. *Syntax:*

```
⟨flowchart⟩ ::= ⟨declaration list⟩; ⟨compound
      tail⟩ . .
  ⟨compound tail⟩ ::= ⟨statement⟩ |
    ⟨statement⟩ ⟨compound tail⟩
```

APPENDIX A

Syntactical Flowchart for BC NELIAC

As an aid in understanding the syntactical rules of BC NELIAC, a flowchart similar to the ALGOL 60 Flowchart has been developed. The shapes of enclosure on the chart have the following meanings:

⬭ Metalinguistic variables appear in ellipses and indicate the enclosed variable is defined at that place on the chart.

▭ Metalinguistic variables appearing in rectangles mean the variable is defined elsewhere on the chart. Grid coordinates for the definition appear at the right of the rectangle.

○ Basic symbols are enclosed in circles.

↓ Vertical arrows indicate a definition of a metalinguistic variable that follows.

→ Horizontal arrows connect the basic symbols and metalinguistic variables which form a definition.

Every metalinguistic formula used to describe BC NELIAC in this report appears on the syntactical flowchart

---

X3.4 Forms ALGOL Task Group

Editor's Note: The following resolution represents the action of X3.4.2 to carry out the directive given to it by X3.4 in the ALGOL resolutions of the April 26, 1963 meeting to consider ALGOL for international standardization and to consider the internal ALGOL effort as a basis for a national standard. —J.G.

At the ASA X3.4 meeting in Detroit on May 20, 1963, the following resolution was presented by X3.4.2 and adopted by X3.4:

"Resolved that a Task Group known as X3.4.2.A will be formed to develop a draft standard on ALGOL and to act as an information exchange for discussion on ALGOL 60, including implementations, within the United States, and corresponding membership without voting membership open to all interested parties."

Mr. Jack Merner was appointed chairman of X3.4.2.A. The following list was suggested as a basis for the formation of X3.4.2.A:

The old ALGOL Working Group
The ALGOL Compiler writers appearing in the *CACM* survey
The United States members of Working Group 2.1
The SHARE ALGOL Committee
The authors of algorithms in the Communications of the ACM
The United States ALGOL authors
COOP Users Group
CUBE Users Group
JUG Users Group

(signed) R. F. CLIPPINGER

APPENDIX B

Transliteration Rules

This appendix represents a summary of equivalences between the character set used with the hardware representation BC NELIAC on the IBM 704 digital computer and the BC NELIAC Reference Language. All of the word delimiters must be separated by blanks in the hardware representation (blanks shown by "_" in hardware representation).

	Character Operator	*Hardware Representation*	*Reference Language Symbols*
	Blank		
	Replacement operator	=	→
	Left arrow	=	←
	Decimal point	.	.
Punctuation operators	Comma	,	,
	Period	.	.
	Semicolon	$	;
Arithmetic operators	Add	+	+
	Subtract	−	−
	Multiply	*	*
	Divide	/	/
Relational operators	Less	_LSS_	<
	Less or equal	_LEQ_	≦
	Equal	_EQU_	=
	Greater or equal	_GEQ_	≧
	Greater	_GTR_	>
	Not equal	_NEQ_	≠
Logical operators	AND	_AND_	∧
	OR	_OR_	∨
Sequential operators	GO TO	_GO_TO_	GO TO
	IF	_IF_	IF
	FOR	_FOR_	FOR
	DO	_DO_	DO
	THEN	_THEN_	THEN
	ELSE	_ELSE_	ELSE
Separator operators	STEP	_STEP_	STEP
	UNTIL	_UNTIL_	UNTIL
	COLON	_CLN_	:
	PERIOD	.	.
	COMMA	,	,
	SEMICOLON	$	;
	Left parenthesis	(	(
	Right parenthesis	)	)
Bracket operators	Left bracket	_LBK_	
	Right bracket	_RBK_	
	BEGIN (equivalent to {)	_BEGIN_ | _LBR_	BEGIN
	END (equivalent to })	_END_ | _RBR_	END
Pseudo operators	Exponentiation	_EXP_	EXP
	Crutch code	_MCH_	MCH
	Octal	_OCT_	OCT
	Alphabetic characters	A . . . Z	A . . . . . Z
	Numeric characters	0 . . . 9	0 9