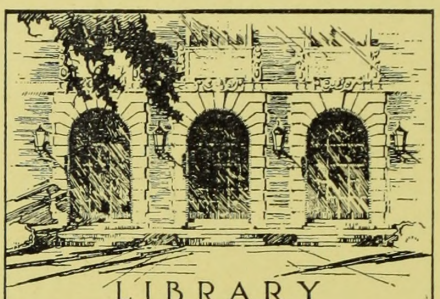
EOL programming examples: a primer, by Leon ukaszewicz and Jurg Nievergelt.

ukaszewicz, Leon.
Urbana, 1967.

https://hdl.handle.net/2027/uiuo.ark:/13960/t2h71z94b



Math

Report No. 242

EOL PROGRAMMING EXAMPLES: A PRIMER

by

Leon Lukaszewicz

and

Jurg Nievergelt

September 1, 1967

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

Report No. 242

EOL PROGRAMMING EXAMPLES: A PRIMER*

by

Leon Lukaszewicz

and

Jurg Nievergelt

September 1, 1967

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. GP-4636.

ACKNOWLEDGEMENT

The development of the EOL language started in 1965 at the Institute for Mathematical Machines in Warsaw, Poland, where the first two versions of the language, EOL-1 and EOL-2, were implemented (see [2] for a description of EOL-2).

The current version EOL-3 was developed and has been implemented on an IBM 7094 at the University of Illinois by Freda S. Fischer, M. Irwin-Zarecki, J. R. Sidlo, D. S. Edgar, and R. Sanders. Their active participation in the project is gratefully acknowledged.

TABLE OF CONTENTS

INTRODUCTION

The purpose of this report is to facilitate the study of the symbol manipulation language EOL, and as such it is a companion to the EOL Report [1].

Three EOL programs or parts of programs, all taken from the general area of compiler writing which we consider to be the main field of application of EOL, are described in detail. The programming techniques used were chosen differently in the three examples in order to emphasize the versatility of EOL.

I. TRANSFORMING ARITHMETIC FORMULAE INTO PARENTHESIS-FREE FORM

Our first example is a complete program called FORMULA which performs the following operations.

(a) Read one formula from the input string I'DATA into the stack S'F.

(b) Transform the formula in S'F into parenthesis-free form.

(c) Write the results on the output Q'RESULT.

It consists of several procedures, each of which is defined as a macro and is used as a macro instruction either in the main procedure FORMULA or in another macro.

1. Problem Specification

1.1. Input

We assume that the input string I'DATA represents an arithmetic formula which satisfies the following conditions:

(a) The formula is of the form
<identifier>::=<arithmetic expression>

(b) Identifiers are composed of letters and digits only.

(c) An arithmetic expression is composed of
-identifiers
-operators : {+|-|*|/}
-parenthesis : {(|)}
-blanks

(d) Each operator or parenthesis can be surrounded by any number of blanks.

(e) A formula is terminated by the character λ, possibly preceded by any number of blanks.

(f) The syntax of arithmetic expressions conforms to the usual rules.

Examples:

Arithmetic formulae of the type described above are

(1) I'DATA: X= ␣ A* ␣(B1+ ␣C) ␣ + ␣ Dλ ...

(2) I'DATA: X1 ␣ = ␣ (A*(C ␣ + ␣ D/Q) ␣ + ␣BA) ␣ * ␣Hλ ...

1.2. Output

We want the output string Q'RESULT to be a sequence of parenthesis-free arithmetic formulae which satisfy the following conditions:

(a) The output formulae taken together are arithmetically equivalent to the input formula.

(b) Each of the output formulae fulfills the conditions a-f required for the input formula, with the exception that they do not contain parentheses.

(c) Each output formula is terminated by the character λ (which is usually interpreted as line-feed carriage return).

(d) New variables which have to be introduced will be denoted by 1R, 2R, 3R, ...

Examples:

The input formulae given in the previous examples will be translated into the following output formulae:

(1) Q'RESULT: 1R=B1+C λ
X=Z*1R+D λ

(2) Q'RESULT: 1R=C+D/Q λ
2R=A*1R+BA λ
X1=2R*H λ

2. Macro Definitions

2.1. The macro READ.ELEMENT

This macro reads a string of characters, called an "element", from input I'X into stack S'Y. All initial blanks in the input string I'X are deleted, and then one of the following operations is performed, depending on the first nonblank character:

(a) If the character belongs to the class R (i.e., is not a blank, letter or digit, but is one of the remaining characters) it is read from I'DATA and put as a constituent at the beginning of S'Y.

(b) If the character belongs to the class LD (i.e., is a letter or a digit), then all the following letters and digits are read up to but excluding the first character of the class BR (i.e., a blank or one of the remaining characters). The string read is put as one constituent at the beginning of S'Y.

Example:

Let

I'X : ␣X1␣=␣31␣␣*␣Cλ␣A ⊣

S'Y : ¯AB ⊣

After executing five times the macro READ.ELEMENT we have:

I'X : λ␣A ⊣

S'Y : ¯C¯*¯31¯=¯X1¯AB ⊣

Macro definition:

```
(1)     READ.ELEMENT : MACRO X, Y
(2)          CLEAR I'X, LDR
(3)          TEST I'X, LD
(4)          GOPL L1
(5)          READ I'X, B'Y, *1
```

```
(6)            RETURN
(7)       L1 : READ I'X, B'Y, BR
(8)            RETURN
(9)            END
```

Remarks:

This macro READ.ELEMENT will be included (in Section 2.3) as a procedure in another macro definition.

The statements in this macro definition have the following meaning:

(1) Macro-heading, which starts the macro and assigns the following names:
   (a) The name READ.ELEMENT to the macro.
   (b) The name X to the input string used in this macro.
   (c) The name Y to the stack used in this macro.

   X and Y are formal parameters of the macro READ.ELEMENT.

(2) Clear initial blanks in I'X. The symbol LDR defines the first character which is not cleared. All characters are divided into four disjoint classes: letters (L), digits (D), blank (B) and all the remaining characters (R). The symbol LDR means: "letter or digit or remaining".

(3) Test if the initial character in I'X is either a letter or a digit. If not, assign "minus" to the variable H.

(4) If H is "plus" go the L1. If H is "minus" change it to "plus" and proceed to the next instruction.

(5) Read one character from the input I'X and put it as a constituent of type word at the end of stack S'Y.

(6) Return to the calling program.

(7) Read from I'X the string of letters and digits, that is, all characters which don't belong to the class "blank or remaining". Put this string as one constituent at the beginning of stack S'Y.

(8) Return to the calling program.

(9) End of the macro definition.

2.2. The macro READ.FORMULA

This macro reads consecutive elements from input string I'DATA and adds them as separate constituents at the end of stack S'F. If the element read is equal to λ it is deleted from S'F and the procedure is terminated. The macro READ.ELEMENT defined previously is used for reading elements.

Example:

Let

I'DATA : ␣ ABC = ␣ ABC+1 ␣ λY=3 ␣ λ ⊣

S'F : ⊣

After executing the macro READ.FORMULA we have:

I'DATA : Y = 3 ␣ λ ⊣

S'F : ¯ABC¯ = ¯ABC¯ + ¯1 ⊣

Macro definition:

```
(1)     READ.FORMULA : MACRO I'DATA, S'F
(2)          READ.ELEMENT : PROC
(3)               READ.ELEMENT DATA, F
(4)          END
(5)          EL : CALL READ.ELEMENT
(6)               TEST B'F, L*
(7)               GOPL L1
(8)               MOVE A'F, Z'F, *1
(9)               GOTO EL
```

```
(10)        L1 : CLEAR A'F, *1
(11)             RETURN
(12)        END
```

Remarks:

The statements in this macro definition have the following meaning:

(1) Macro-heading, which starts the macro and assigns the following names:

(a) The name of the macro, expressed as a sequence of three symbols: READ.FORMULA, I, S.

(b) The name DATA to the input string used in the macro.

(c) The name F to the stack used in the macro.

The symbols I and S on the right side of the word MACRO are called "macro name continuations". They may be used to distinguish between possibly many macros whose name begins with READ.FORMULA.

(2) Procedure-heading, which indicates that the macro READ.FORMULA contains a procedure called READ.ELEMENT. So far, this procedure READ.ELEMENT has nothing to do with the macro READ.ELEMENT defined earlier.

(3) A macro instruction, which causes the EOL compiler to insert a copy of the previously defined macro READ.ELEMENT at this point. The formal parameters X and Y of the macro definition are replaced by the actual parameters DATA and F, respectively. The EOL compiler recognizes this to be a macro instruction because READ.ELEMENT is not one of the standard EOL keywords (see Section III of this report), and is not a label (as it was in (2) above) because it is not followed by a colon.

(4) End of the procedure READ.ELEMENT, which, by virtue of the macro instruction (3), has as its procedure body the macro called READ.ELEMENT (with parameters replaced as described in (3) above).

(5) Calls for execution of the procedure READ.ELEMENT, which reads one element from I'DATA and puts it at the top of stack S'F.

(6) Tests to see if the element just read is a λ. If the test is positive the value of H is unchanged, that is, it remains "plus". If the test fails, the value of H is set to "minus".

(7) If the value of H is "plus" go to statement (8). Otherwise reset H to "plus" and proceed to the next statement.

(8) Move the first constituent of S'F (it is different from $\bar{\lambda}$) to the end of S'F.

(9) Read further elements until a λ is found.

(10) Clear the initial constituent in stack S'F (it is equal to $\bar{\lambda}$).

(11) Return to the calling program.

(12) End of the macro definition.

2.3. Including the macro READ.FORMULA

For including into a program the macro READ.FORMULA defined in Section 2.2., the following statement can be used:

(1) READ.FORMULA I'7, S'5

This macro instruction is replaced during compile time by the macro-definition READ.FORMULA in which formal parameters DATA and F are replaced by actual parameters 7 and 5, respectively. Note that the macro READ.FORMULA contains the statement

READ.ELEMENT DATA, F

which causes the inclusion of the macro READ.ELEMENT. The body of the latter macro is inserted at compile time in place of the macro instruction above, with the values of X and Y equal to the actual values of DATA and F, i.e., to 7 and 5, respectively.

The final result of statement (1) is the following:

```
READ.ELEMENT:PROC
        CLEAR I7, LDR
        TEST I7, LD
        GOPL L1
        READ I7, B5, *1
        RETURN
    L1:READ I7, B5, RB
        RETURN
        END

    EL:CALL READ.ELEMENT
        TEST B5, L*
        GOPL L1
        MOVE A5, Z5, *1
        GOTO EL
    L1:CLEAR A5, *1
        RETURN
```

Notice that because of the block structure of EOL the two declarations of the label L1 do not cause this label to be multiply defined. The two declarations have different scopes (see Section II of this report).

2.4. The macro GENERATOR

This macro assumes that the initial constituent in the stack S'MODEL is a word ¯N, composed of digits only, and the following constituent a word ¯W. The macro GENERATOR forms one word of the structure.

¯NW

and adds this word at the beginning of the stack S'NEW.VAR. Then the word on the top of stack S'MODEL is changed in such a way that it represents the original number increased by one.

Example:

Let

S'MODEL : ¯3¯A¯BX ⊣

S'NEW.VAR : ¯Q ⊣

After executing the macro GENERATOR twice we have:

S'MODEL : ¯5¯A¯BX ⊣

S'NEW.VAR : ¯4A¯3A¯Q ⊣

Macro definition:

```
(1)     GENERATOR:MACRO S'MODEL, S'NEW.VAR
(2)          MOVE B'MODEL, B'NEW.VAR, *2
(3)          COMPRESS A'NEW.VAR,,*2
(4)          NUMBER A'MODEL
(5)          ADD A'MODEL,,1
(6)          WORD A'MODEL
(7)          END
```

Remarks:

Let us assume that the intial values of S'MODEL and S'NEW.VAR are the same as the ones given in the example in this section. Then the statements of the procedure GENERATOR listed below act as follows:

(1) Macro heading.

(2) The initial two constituents of S'MODEL are moved to the beginning of S'NEW.VAR, so that

S'NEW.VAR : ¯3¯A¯Q ⊣

The value of S'MODEL remains unchanged.

(3) The initial two constituents of S'NEW.VAR are compressed into one constituent: S'NEW.VAR : ¯3A¯Q ⊣
Notice that the missing second argument of this instruction has the same effect as if it were present and equal to B'NEW.VAR.

(4) The initial constituent of S'MODEL, which we assume is a string of digits, is converted from word to number form: S'MODEL : °3¯A¯BX ⊣
In (4), (5) and (6) the missing second argument has the same effect as if it were present and equal to B'MODEL.

(5) The number on the top of S'MODEL is increased by one: S'MODEL : °4¯A¯BX ⊣

(6) The initial constituent of S'MODEL is converted back to word from: S'MODEL : ¯4¯A¯BX ⊣

(7) End of macro definition.

If the macro GENERATOR is executed again, the result would be the one indicated in the example given at the beginning of this section.

2.5. The macro PARENTHESIS

It is assumed that stack S'F contains an arithmetic formula, read in from I'DATA by the macro READ.FORMULA, that fulfills the conditions specified in Section 1.1. The procedure PARENTHESIS transforms this formula into an equivalent sequence of parenthesis-free formulae, separated by constituents ¯λ, which are put back on stack S'F.

We further assume that prior to execution of the macro PARENTHESIS the stacks W1 and W2 are empty and that the two initial constituents of the stack S'MODEL satisfy the conditions stated in Section 2.4.

Example:

Let us assume that

S'F : ¯X1¯=¯(¯A¯*¯(¯C¯+¯D¯/¯Q¯)¯+¯BA¯)¯*¯H ⊣

S'W1 : ⊣

S'W2 : ⊣

S'MODEL : ¯1¯R ⊣

H : +

Then, after execution of the macro PARENTHESIS the content of S'F is equal to

S'F : ¯1R¯=¯C¯+¯D¯/¯Q¯λ¯2R¯=¯A¯*¯1R¯+¯BA¯λ¯X1¯=¯2R¯*¯H¯λ ⊣

Macro definition:

```
(1)     PARENTHESIS : MACRO S'F, S'MODEL, W1, W2
(2)     P : MOVE A'F, A'W2, ')'
(3)         GOMI END
(4)         CLEAR A'F, *1
(5)         GENERATOR S'MODEL, S'F
(6)         MOVE B'F, Z'W1, *1
(7)         SET Z'W1, '='
(8)         MOVE A'W2, Y'W1, '('
(9)         CLEAR A'W2, *1
(10)        SET Z'W1, L*
(11)        GOTO P
(12)    END : SET A'F, L*
(13)        MOVE A'W2, A'F
(14)        MOVE A'W1, B'F
(15)        RETURN
(16)        END
```

Remarks:

Let us assume that

S'F : ¯X¯=¯A¯*¯(¯B1¯+¯C¯)¯+¯D ⊣

S'W1 : ⊣

S'W2 : ⊣

S'MODEL : ¯1¯R ⊣

H : +

Then, execution of the macro PARENTHESIS proceeds in the order listed below with the effects indicated. (1) is the macro heading and (5) causes the inclusion (at compile time) of the macro GENERATOR defined in Section 2.4.

(2) S'F : ¯)¯+¯D ⊣
S'W2 : ¯C¯+¯B1¯(¯*¯A¯=¯X ⊣
H : +

(3) Go to the next instruction.

(4) S'F : ¯+¯D ⊣

(5) S'F : ¯1R¯+¯D ⊣
S'MODEL: ¯2¯R ⊣

(6) S'W1 : ¯1R ⊣

(7) S'W1 : ¯1R¯= ⊣

(8) S'W1 : ¯1R¯=¯B1¯+¯C ⊣
S'W2 : ¯(¯*¯A¯=¯X ⊣

(9) S'W2 : ¯*¯A¯=¯X ⊣

(10) S'W1 : ¯1R¯=¯B1¯+¯C¯λ ⊣

(11) At this point the program has removed one pair of parenthesis and is ready to look for the next pair.

(2) S'F : ⊣
S'W2 : ¯D¯+¯1R¯*¯A¯=¯X1 ⊣
H : -

```
(3)       Go to END and reset H to "plus".
(12)      S'F   : λ
(13)      S'W2  : ⊣
          S'F   : X1 = A * 1R + D λ
(14)      S'W1  : ⊣
          S'F   : 1R = B1 + C λ X1 = A * 1R + D λ
(15)      Return to the calling program.
```

2.6. The macro TRANSFORM

This macro transforms the input formula into the equivalent sequence of parenthesis-free output formulae according to the problem specification given in Section 1. We assume that prior to the execution of the macro TRANSFORM the stacks W1 and W2 are empty.

Procedure:

```
(1)       TRANSFORM:MACRO I'DATA, Q'RESULT, S'F, S'MOD, S'W1, S'W2
          ┌ READ.FORMULA:PROC
(2)       │     READ.FORMULA I'DATA, S'F
          └ END
          ┌ PARENTHESIS:PROC
(3)       │     PARENTHESIS S'F, S'MOD, W1, W2
          └ END
(4)           SET B'MOD, 'R'
(5)           SET B'MOD, '1'
(6)           CALL READ.FORMULA
(7)           CALL PARENTHESIS
```

```
(8)          WRITE A'F, Q'RESULT
(9)      END
```

Remarks:

(1) Macro heading.

(2) Include the macro READ.FORMULA as a procedure.

(3) Include the macro PARENTHESIS as a procedure.

(4) (5) Set the first two constituents of stack S'MODEL equal to ¯1 and ¯R, respectively.

(6) Execute the procedure READ.FORMULA.

(7) Execute the procedure PARENTHESIS.

(8) Print out the result.

(9) End of macro definition.

3. The Main Procedure FORMULA

The action of the main procedure FORMULA is merely to execute the macro TRANSFORM. The complete program, which includes all macro-definitions, is shown and described below.

Complete Program:

```
(1)  READ.ELEMENT:MACRO X, Y
                  ------
                  ------

(2)  READ.FORMULA:MACRO I'DATA, S'F
                  ------
                  ------

(3)  GENERATOR:MACRO S'MODEL, S'NEW.VAR
                  ------
                  ------

(4)  PARENTHESIS:MACRO S'F, S'MODEL, W1, W2
                  ------
                  ------
```

(5) TRANSFORM:MACRO I'DATA, Q'RESULT

---

(6) FORMULA:MAIN

(7) TRANSFORM I'7, Q'6, S'1, S'2, S'30, S'31

(8) STOP

(9) END

Remarks:

(1) through (5): Macro definitions

(6) Heading of main procedure

(7) This macro instruction causes the insertion, at compile time, of the procedure TRANSFORM with actual parameters 7, 6, 1, 2, 30, 31 in place of the formal parameters DATA, RESULT, F, MOD, W1, W2. Since TRANSFORM includes PARENTHESIS, which in turn includes GENERATOR, etc, all the macros (1) through (5) are assembled in the proper place to make up the main procedure.

Execution of the main procedure FORMULA begins with the first instruction of the macro TRANSFORM, i.e. instruction (4) of Section 2.6.

(8) Terminate execution.

(9) End of main procedure FORMULA.

II. GENERATING TABLES OF IDENTIFIERS FOR PROGRAMS WITH NESTED BLOCK STRUCTURE

In a programming language which allows nested block structure, such as ALGOL, PL/1, EOL, there are two special symbols, say BEGIN and END, which occur in programs in the same way that left and right parentheses occur in arithmetic expressions. The main purpose of block structure is to delimit the scope of identifiers (labels, names of variables, etc.) to those parts of a program where they are used, thus allowing the use of the same identifier to denote different quantities in different parts of a program.

A block is defined by the occurrence of a symbol BEGIN and the corresponding occurrence of the symbol END.

The content of a block B consists of all parts of the program located between the BEGIN and the corresponding END which define B. If block B is located in the content of block A, then B is a subblock of A. The interior of a block B consists of the content of B with the exception of the contents of all subblocks of B.

An identifier may be declared in a program in several places and in several ways; e.g. by using it as the label of a statement, by declaring it to be a variable of a certain type, or implicitly by simply naming a variable with this particular identifier.

For each declaration of an identifier in a program with nested block structure, a certain part of this program is defined as the scope of this declaration. In this part any occurrence of this identifier denotes the quantity referred to in the declaration. If the same identifier I is declared in exactly two different blocks, use of I in the program will denote: the quantity defined by the first declaration anywhere within the scope of the first declaration; the quantity defined by the second declaration anywhere within the scope of the second declaration; and will be an undefined name outside the scope of either declaration.

The scope of a declaration D of an identifier I is defined to be the content of that block B in whose interior D occurs , with the exception of the contents of all subblocks of B in whose interior another declaration D' of the same identifier I occurs.

Figure 1 shows a block structure nested to a depth of 4. Three declarations of the same identifier I occur in the interior of the blocks labelled B, C, D, respectively. Since C and D are subblocks of B, the scope of the declaration of I in the interior of block B is interrupted by the scopes of the other two declarations.

Any compiler must generate a table of all identifiers used in a program and their associated descriptors (e.g. the address of the memory location that the compiler has assigned to this identifier). A compiler for a block structured programming language in addition must store in the table information concerning the scope of each declaration of an identifier.

The method described below generates such a table in a first pass (left to right scan) of the program to be compiled, and in a second pass inserts the address of the memory location assigned to this identifier in each place in the program where this identifier is used.

The table is segmented into sections, one section for the interior of each block of the program to be compiled. Thus in each section an identifier may occur at most once, while the same identifier may occur in different sections (in which case it denotes a different quantity in each section). As the program to be compiled is scanned (from left to right) in the first pass, each BEGIN causes a new section of the table to be started, and the address of the previous section is saved on a pushdown stack. Then all identifiers declared in the new block are entered into this new section of the table with their descriptors.

Each time an END is encountered, the section last operated upon is permanently closed, and the section whose address is on top of the stack is reopened.

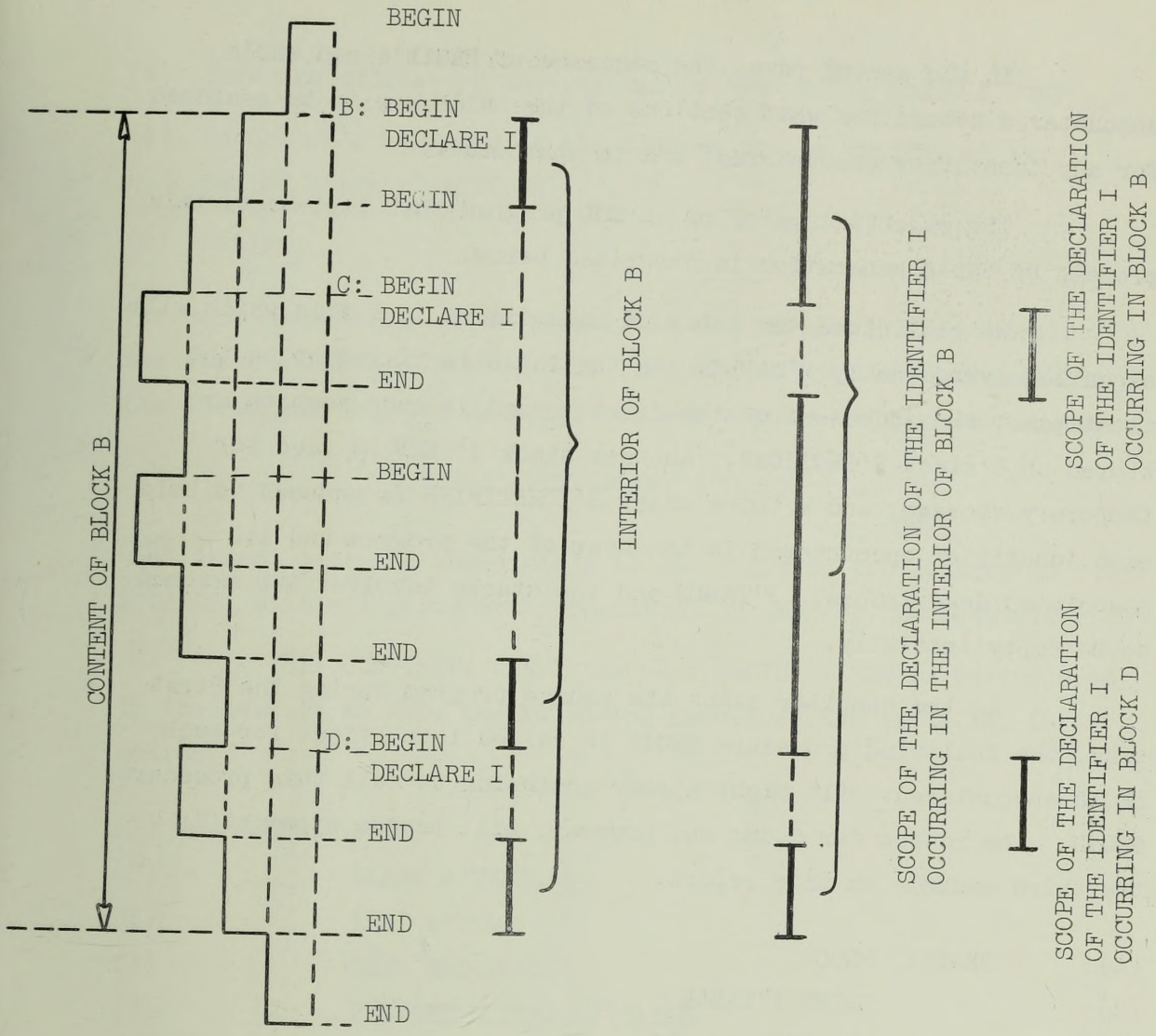


Figure 1

In the second pass, the sequence of BEGIN's and END's encountered determines what sections of the table are to be searched for any identifier encountered, and in what order.

The essential parts of an EOL program that implements this process of table generation is described below.

We will store the table of identifiers in a file which will be called symbolically P'TABLE. As the table is filled up on the first pass, the addresses of the first record in each section are stored on a stack S'SECTIONS. Another stack S'TEMP is used for temporary storage, and a third stack S'IDENTIFIER is assumed to hold each identifier encountered in the scan of the program and its associated descriptors. P'TABLE and the stacks involved are assumed to be empty initially.

As the compiler scans the source program during the first pass, the following procedure BEGIN is called into effect for each BEGIN encountered. (It might appear confusing to call this procedure BEGIN. The reason for doing so, however, will become apparent in the third example in this report.)

```
(1)     BEGIN:  PROC
(2)             SHIFT P'TABLE
(3)             SET D'TABLE, S*
(4)             SAVE P'TABLE, B'TEMP
(5)             MØVE B'TEMP, Z'SECTIONS, *1
(6)             RETURN
                END
```

The procedure BEGIN performs the following operations:

(1) (Procedure heading)

(2) Shift the file pointer to the rightmost end of file P'TABLE

(3) Add a special record $\check{}\sigma$ to the left of the pointer to denote the beginning of a new section

(4) Save the address of the pointer on top of the stack S'TEMP

(5) Establish a new copy of the address of the pointer at the end of stack S'SECTIONS

(6) Return

For each declaration of an identifier encountered during compilation the stack S'IDENTIFIER will be filled with constituents representing the identifier and its descriptors, and then the single instruction

PUT A'IDENTIFIER, C'TABLE

will insert a record immediately to the right of the pointer in file P'TABLE, with all pertinent information concerning the identifier.

This continues until the next BEGIN or END is encountered. In the case of an END, the following procedure END is called into action.

```
(1)     END:  PROC
(2)           CLEAR A'TEMP,*1
(3)           TEST B'TEMP
(4)           GØPL END.PASS.1
(5)           RESTORE B'TEMP, P'TABLE
(6)           RETURN
              END
```

This procedure performs the following action:

(2) Clear the first constituent of stack S'TEMP, namely the address of the section last operated upon in P'TABLE, since this section is permanently closed.

(3) and (4) If S'TEMP is now empty, the last END of the program has been detected, in which case control should be transferred to label END.PASS.1, whose name speaks for itself.

(5) Otherwise, restore the pointer in file P'TABLE to the beginning of that section which was active just before the section terminated by the END being scanned.

In Figure 3, the contents of the stacks and of the file P'TABLE are shown just after each BEGIN and END of a program with the block structure shown in Figure 2 has been processed.

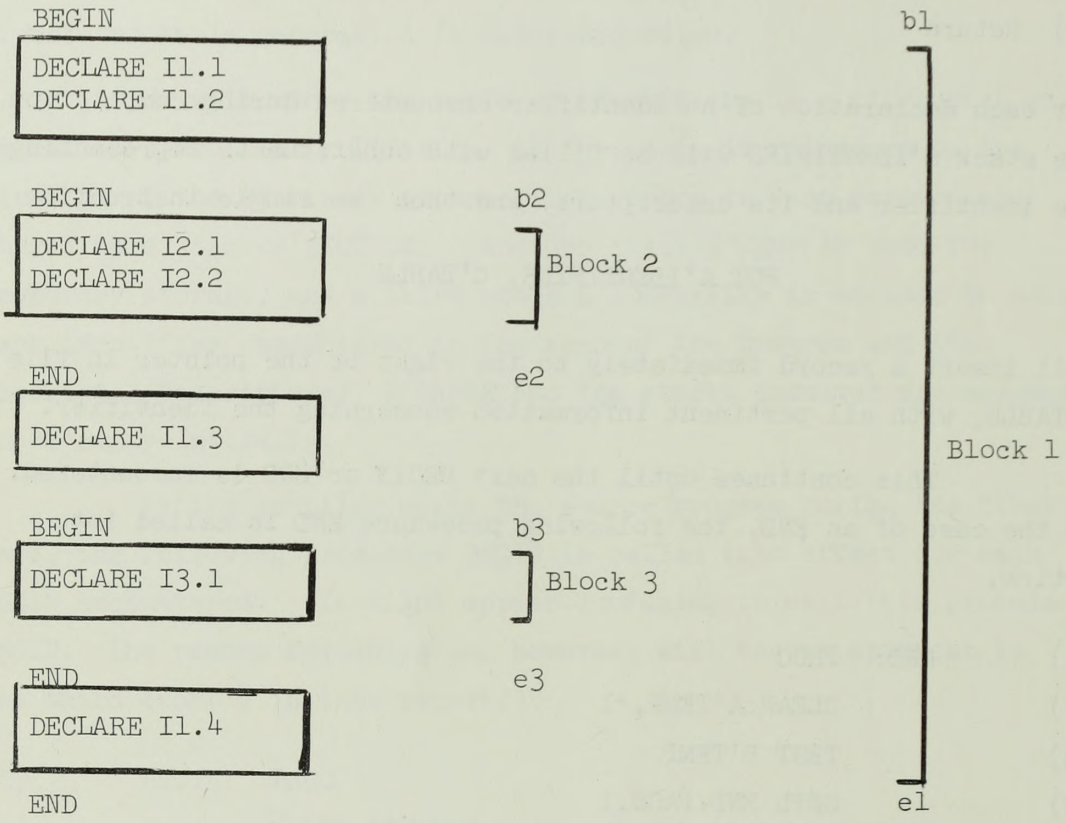
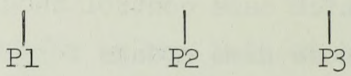


Figure 2

The records associated with identifiers I1.1, I1.2, etc. are denoted by R1.1, R1.2, etc. The pointer in the file P'TABLE is denoted by ↑, addresses of records by P1, P2, P3, and the corresponding positions in the file (if they don't correspond with the position of the file pointer) by



During the second pass, searching appropriate sections of P'TABLE to find the record which corresponds to a given identifier is governed by the content of S'TEMP. The latter in turn is governed by the sequence of BEGIN's and END's in the program to be processed according to the following rules:

Whenever a BEGIN is encountered, the instruction

MOVE A'SECTIONS, B'TEMP,*1

moves (and deletes) the first constituent from S'SECTIONS to the beginning of S'TEMP.

Whenever an END is encountered, the instruction

CLEAR A'TEMP,*1

deletes the first constituent of S'TEMP.

This has the effect that, as one is scanning the program to be processed, the top of stack S'TEMP always contains the address of that section P'TABLE which corresponds to the interior of the block now being scanned. The second constituent of S'TEMP points to the section of P'TABLE which corresponds to the interior of the block which immediately surrounds the block whose interior is being scanned. In general, the constituents of S'TEMP, from first to last, point to sections of P'TABLE which correspond to the block whose interior is being scanned, the immediately surrounding block, etc., up to the outermost block.

Thus, a search for the record corresponding to a given identifier begins in the section addressed by the first constituent of S'TEMP. If the record is not found there, the search proceeds to the section addressed by the second constituent, and so on. If not found by the time the section of P'TABLE addressed by the last constituent of S'TEMP has been searched, the identifier in question must be undefined.

S'SECTIONS : ⊣

S'TEMP : ⊣

P'TABLE : ↑ ⊣

---

b1) S'SECTIONS : $\overset{\wedge}{P1}$ ⊣

S'TEMP : $\overset{\wedge}{P1}$ ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma\uparrow}}$ ⊣

---

b2) S'SECTIONS : $\overset{\wedge}{P1}$ $\overset{\wedge}{P2}$ ⊣

S'TEMP : $\overset{\wedge}{P2}$ $\overset{\wedge}{P1}$ ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma|}}$ $\vee$R1.2 $\vee$R1.1 $\underset{P2}{\overset{\vee}{\sigma\uparrow}}$ ⊣

---

e2) S'SECTIONS : $\overset{\wedge}{P1}$ $\overset{\wedge}{P2}$ ⊣

S'TEMP : $\overset{\wedge}{P1}$ ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma\uparrow}}$ $\vee$R1.2 $\vee$R1.1 $\underset{P2}{\overset{\vee}{\sigma|}}$ $\vee$R2.2 $\vee$R2.1 ⊣

---

b3) S'SECTIONS : $\overset{\wedge}{P1}$ $\overset{\wedge}{P2}$ $\overset{\wedge}{P3}$ ⊣

S'TEMP : $\overset{\wedge}{P3}$ $\overset{\wedge}{P1}$ ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma|}}$ $\vee$R1.3 $\vee$R1.2 $\vee$R1.1 $\underset{P2}{\overset{\vee}{\sigma|}}$ $\vee$R2.2 $\vee$R2.1 $\underset{P3}{\overset{\vee}{\sigma\uparrow}}$ ⊣

---

e3) S'SECTIONS : $\overset{\wedge}{P1}$ $\overset{\wedge}{P2}$ $\overset{\wedge}{P3}$ ⊣

S'TEMP : $\overset{\wedge}{P1}$ ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma\uparrow}}$ $\vee$R1.3 $\vee$R1.2 $\vee$R1.1 $\underset{P2}{\overset{\vee}{\sigma|}}$ $\vee$R2.2 $\vee$R2.1 $\underset{P3}{\overset{\vee}{\sigma|}}$ R3.1 ⊣

---

e1) S'SECTIONS : $\overset{\wedge}{P1}$ $\overset{\wedge}{P2}$ $\overset{\wedge}{P3}$ ⊣

S'TEMP : ⊣

P'TABLE : $\underset{P1}{\overset{\vee}{\sigma\uparrow}}$ $\vee$R1.4 $\vee$R1.3 $\vee$R1.2 $\vee$R1.1 $\underset{P2}{\overset{\vee}{\sigma|}}$ $\vee$R2.2 $\vee$R2.1 $\underset{P3}{\overset{\vee}{\sigma|}}$ R3.1 ⊣

Figure 3

The contents of S'SECTIONS and S'TEMP just after each BEGIN and END of the same program (Figure 2) has been processed are shown in Figure 4.

Processing of programs written in any language which allows nested block structures always calls for pushdown stacks. The example discussed is representative of the ease and conciseness with which such operations can be expressed in EOL.

```
      S'SECTIONS  :  ^P1 ^P2 ^P3 ⊣
      S'TEMP      :  ⊣
```

```
b1)   S'SECTIONS  :  ^P2 ^P3 ⊣
      S'TEMP      :  ^P1 ⊣
```

```
b2)   S'SECTIONS  :  ^P3 ⊣
      S'TEMP      :  ^P2 ^P1 ⊣
```

```
e2)   S'SECTIONS  :  ^P3 ⊣
      S'TEMP      :  ^P1 ⊣
```

```
b3)   S'SECTIONS  :  ⊣
      S'TEMP      :  ^P3 ^P1 ⊣
```

```
e3)   S'SECTIONS  :  ⊣
      S'TEMP      :  ^P1 ⊣
```

```
e1)   S'SECTIONS  :  ⊣
      S'TEMP      :  ⊣
```

Figure 4

III. THE USE OF SWITCHES IN A COMPILER

This example describes one feature of the implementation of the EOL language on a computer.

An EOL program (called source program below) is first compiled by the EOL compiler, which is a program also written in EOL, into an internal representation. In this representation most instructions of the EOL source program are represented by the contents of one word in the computer memory (e.g. 36 bits).

The compilation proceeds in two passes. The first pass reads all macro definitions (which have to appear at the beginning of the program), inserts these in place of the corresponding macro instructions, generates tables of labels and formal indexes (see example II of this report), and replaces the keyword and certain other parts of an EOL line (see explanation below) by numerical codes. The second pass replaces all labels and formal indexes used by their actual value (addresses or actual indexes).

The internal representation of an EOL source program is then interpreted by a program (written in machine language) called the EOL interpreter. This process of interpretation is also called "execution of the EOL program".

This example outlines a procedure called READ.LABEL+KEY of the first pass of the EOL compiler which makes essential use of a name switch.

An EOL program can be thought of as consisting of a sequence of "lines", each one separated from the next by an occurrence of the special character λ. A line may be:

(1) An instruction, possibly labelled, e.g.:

```
L : MOVE ␣ A1, B2, *1
    M ␣ S'4
```

The second is an example of a macro instruction. The macro definition to which it refers might have a heading as shown in the example (3) below.

(2) A declaration, e.g.:

X : EQU ␣ 5

Z : ENTRY

T : TABLE ␣ 'A', 'B', 'C'

(3) A procedure heading or a macro heading, e.g.:

P : PROC

M : MACRO ␣ S'PARAMETER

(4) A procedure or a macro end, i.e.:

END

(5) A comment, e.g.:

COMMENT ␣ A COMMENT STRING MAY NOT BEGIN WITH A COLON

The general form of an EOL line is

[<label>:]..<key>[␣ <rest>] λ

which means that:

(a) The line begins with an arbitrary number of labels (possibly none), separated from each other and from the keyword by a colon ":".

(b) There follows the keyword of the line, which is either: one of about 50 standard EOL keywords (e.g. MOVE, EQU, PROC, COMMENT), or else: the name of a user-defined macro.

(c) Between the keyword and the terminal character λ, there may be other parts of the line, such as arguments of an instruction or the comment string of a comment statement. If such a part is present, it must be preceded by a blank.

The EOL compiler in processing an EOL line decides on what action to take mainly as a function of the keyword. Hence it has a procedure, called READ.LABEL+KEY, which reads an EOL line from left to right, stores all the labels (if any) occuring on the left of the key on a stack S'LK, reads the keyword and uses it to branch through a name switch to one of a large number of places in the program which processes EOL lines with this particular key.

To understand how the procedure READ.LABEL+KEY identifies labels and the keyword, one has to remember the following:

(d) Labels and keywords may not contain blanks, primes or colons.

(e) The "rest" of an EOL line (i.e. arguments, comment strings, etc.) may not begin with a colon.

(f) In the usual representation of the EOL reference language, there may occur an arbitrary number of blanks on either side of a delimiter, and the symbol " ␣ " (blank) may be expanded into an arbitrary non zero number of blanks. For our purpose this means that an EOL line may begin with an arbitrary number of blanks, every colon may be surrounded by an arbitrary number of blanks, and the keyword may be separated from the rest of the line by one or more blanks.

An example of an EOL line is:

␣␣L1␣␣:␣L2:GOTO␣␣␣L3␣λ

↑ this blank is mandatory

The procedure READ.LABEL+KEY, after having read from left to right up to (but excluding) a blank or a colon, decides on whether what it has read is a label or the keyword on the basis of the first nonblank character that follows: if it is a colon, a label was read, otherwise it was the keyword.

Parts of this procedure of the EOL compiler are described below. It has been simplified further by omitting all parts which check the syntax of the EOL line to be processed (i.e. which will detect illegal EOL lines, such as those which have no keyword).

```
READ.LABEL+KEY : PROC
     PROGRAM   : EQU 7
     COMMENT  THE LINE TO BE PROCESSED WILL BE
     C        READ FROM INPUT I7
     LK : EQU 1
     CLEAR A'LK
     COMMENT  STACK S'LK WILL HOLD ALL THE
     C        LABELS FOUND ON THE LEFT OF THE KEY-
     C        WORD, ONE CONSTITUENT PER LABEL,
     C        AND THE KEYWORD AS THE FIRST
     C        CONSTITUENT
     TESTCHR : EQU 2
     SET  B'TESTCHR, ':'
     SET  B'TESTCHR, '␣'
     COMMENT  STACK S'TESTCHR HOLDS TWO CONSTANT
     C        CONSTITUENTS, COLON AND BLANK,
     C        USED TO FACILITATE TESTING THE
     C        INPUT STRING
READ : CLEAR I'PROGRAM, LDR
     COMMENT  CLEAR INITIAL BLANKS
     READ  I'PROGRAM, B'LK, T'TESTCHR
     COMMENT  READ INPUT STRING UP TO BUT
     C        EXCLUDING THE FIRST BLANK OR COLON
     C        AND PUT IT AS ONE CONSTITUENT
     C        AT THE BEGINNING OF STACK LK
```

```
      TEST  I'PROGRAM, B
      COMMENT  IS THE NEXT CHARACTER A BLANK
      GOMI     COLON
      COMMENT  IF NOT, IT MUST BE A COLON
      CLEAR  I'PROGRAM, LDR
      COMMENT  IF YES, CLEAR ALL CONSECUTIVE BLANKS
      TEST  I'PROGRAM, ':'
      COMMENT  IS THE NEXT CHARACTER A COLON
      GOMI     KEY
      COMMENT  IF NOT, WE MUST HAVE READ THE KEYWORD
      C        IF YES, WE MUST HAVE READ A LABEL
COLON : CLEAR I'PROGRAM, *1
      COMMENT  DISCARD THE COLON, AND WE ARE
      C        READY TO READ ANOTHER LABEL
      C        OR THE KEYWORD
      GOTO     READ
KEY   : CANA B'LK, SW.KEY
      COMMENT  USE THE KEYWORD JUST READ
      C        TO BRANCH THRU THE NAME SWITCH
      C        SW.KEY.
      C        IF THE KEYWORD IS NOT ON THE
      C        LABEL LIST OF THE SWITCH, IT MUST
      C        BE THE NAME OF A MACRO.  IN THIS CASE,
      C        THE SWITCHJUMP IS NOT EXECUTED, H IS
      C        SET TO MINUS, AND CONTROL PASSES TO
      C        THE NEXT INSTRUCTION BELOW.  THE VARIOUS
      C        PROCEDURES THAT ARE CALLED THRU
      C        THE SWITCH SW.KEY ALSO RETURN
      C        CONTROL TO THE NEXT INSTRUCTION BELOW,
      C        BUT IN THEIR CASE H IS PLUS
      GOMI     MACRO
      COMMENT  IF H WAS MINUS, PROCESS MACRO
      C        INSTRUCTION.  THIS PART OF THE PROGRAM
      C        IS NOT INCLUDED IN THIS EXAMPLE.
```

```
      C         IF H WAS PLUS, ON THE OTHER HAND,
      C         CONTROL HAS JUST RETURNED FROM
      C         ONE OF THE PROCEDURES DESCRIBED
      C         BELOW, AND READ.LABEL+KEY IS
      C         FINISHED.  HENCE
      RETURN
      COMMENT   JUST BELOW IS THE BIG NAME SWITCH
      C         WHOSE LABEL LIST CONSISTS OF ALL
      C         EOL KEYWORDS.  THE MINUS SIGNS
      C         ON THE RIGHT ARE CONTINUATION MARKS,
      C         WHICH INDICATE THAT THE WHOLE
      C         SWITCH DECLARATION IS ONE EOL LINE
SW.KEY : NAME ADD, BLANK, CAIN, CALL, CANA            -
              CAPL, CLEAR, CODE, COMMENT              -
                 .
                 .
                 .
                     ..., WORD, WRITE
      COMMENT   THE REST OF THE PROCEDURE
      C         READ.LABEL+KEY IS MADE UP OF
      C         PROCEDURES THAT PROCESS EOL LINES
      C         WITH VARIOUS KEYWORDS.  TWO OF
      C         THESE MIGHT BE THE PROCEDURES
      C         'BEGIN' AND 'END' DESCRIBED IN
      C         SECTION II OF THIS REPORT.  TWO
      C         OTHERS ARE MENTIONED BELOW:
ADD:SUB:MULT:DIV:PROC
              +-------------------------------+
              | process arithmetic instructions |
              +-------------------------------+
              RETURN
              END
      .
      .
      .
CAIN:CANA:GOIN:GONA:PROC
              +-------------------------+
              | process switch jumps    |
              +-------------------------+
```

```
          RETURN
          END
END OF PROCEDURE READ.LABEL+KEY
COMMENT   EVERYTHING TO THE RIGHT OF
C         KEYWORD 'END' ABOVE IS A
C         COMMENT STRING.
```

BIBLIOGRAPHY

[1] Lukaszewicz, L. and Nievergelt, J., "EOL Report," Report No. 241, Department of Computer Science, University of Illinois, Urbana, Illinois; September 1967.

[2] Lukaszewicz, L., "EOL-A Symbol Manipulation Language," The Computer Journal, Vol. 10, No. 1, pp.53-59; May 1967.