EOL—a symbol manipulation language

By L. Lukaszewicz*

A simple symbol manipulation language is presented. The language has been designed mainly as a tool for implementing automatic programming systems. It permits of an arbitrary form of input and output strings and is equipped with separate devices for processing short but complex expressions and for files containing a great amount of information. The instruction format is simple, and more complex operations can be defined by means of procedures, which can be written in the machine language. The possibility of modifying a program before its performance is provided.

EOL is a *simple*, low level language for manipulating symbols expressed as strings of characters. The field of EOL application is rather wide, although the language has been designed mainly as a tool for implementing automatic programming systems.

The original application of EOL was the processing of arithmetic and logic expressions, hence it has been named Expression-Oriented Language. Symbolic differentiation of a function is an example of such an application. The function is given in the form of a formula, and the formula of the derivative of this function is to be found. This task is not very difficult, as the rules of function differentiation are relatively simple. It does not require many numerical computations, apart from simple integer arithmetic operations, but it does require a certain amount of manipulation on symbols constituting the initial formula.

Another application of the symbol manipulation language is to determine a function defined as a definite integral with a variable integration limit. The simplest way of solution is to present the result as the difference between primitive functions. However, the rules defining the primitive functions are rather complex and they do not always lead to a result, since very often these functions cannot be expressed as finite expressions, composed of elementary functions only. Because of this, the program begins with an attempt to solve the problem by defining the primitive function by a method relying mainly on symbol manipulation. If, however, no results are obtained in this way, the problem is solved using one of the numerical integration routines. Thus, in this example, the general method of solution is chosen by the program, reducing the programmer's work to a minimum. In particular he is freed from searching for the primitive function of the integrand.

Generally, the basic aim of problem-oriented languages is to free the programmer from forming procedures leading to the solution of the problem. One of the basic EOL aims is to make the implementation of problem-oriented languages easier.

A well-known example of a problem-oriented language is a report generator, commonly used in data-processing problems. However, there exists a possibility of creating other problem-oriented languages in many fields of science and technology and the EOL system has been used for these purposes.

An example of such a language may be the problem-oriented language for solving ordinary differential equations. In this case the programmer writes the problem specification in the form of a set of equations and initial conditions, and establishes the required precision and format of the results. The task is to transform the string of symbols representing the problem specification into a string of symbols representing the appropriate program for the actual solution of this problem in e.g. ALGOL.

An example of a still more specialized problem-oriented language might be a language for electric network analysis aiming to reduce the programmer's work to a minimum. With this system the programmer starts with the network description by writing down all the node symbols, and the characteristics of the branches connecting these nodes. Then he determines the currents or voltages, which are of interest to him, and establishes the form of the results. The task of the system is to transform the so-written problem specification into a set of differential equations describing this network, and then to build the program which would solve it in the required output format.

This task provides a range of non-trivial problems, as the system must carry out a proper network analysis. For instance, the system has to decide when the first and when the second Kirchoff's law should be applied, possibly find a system of independent network lattices, and finally select the set of equations so as to obtain the simplest form of results. In such a case, the whole method of the electric network analysis should be contained in a problem-oriented system and this method might even be unknown to the programmer.

In general, the level of automation in problem-oriented languages is high and the programmer is required only to provide a necessary minimum of information about the problem he is interested in. This information is often reduced to a concise but complete description of the system being investigated and to the specification of desired outputs. On this basis the

* *Instytut Maszyn Matematycznych, Warsaw, Poland.*

system chooses appropriate procedures for solving the problem.

The implementation of universal procedure-oriented languages of automatic programming, such as ALGOL and COBOL, frequently consists in translating programs written in procedure-oriented language into programs written in machine language which use symbolic notation. As all these programs are written in the form of strings of symbols, the processing presents a problem belonging to the field of symbol manipulation. As procedure-oriented languages are used very often, their translation into machine language should be efficient; for instance, the translation time should be reasonably short. These matters have also been taken into account while designing the EOL language.

The development of the EOL language started two years ago at the Institute of Mathematical Machines in Warsaw. The initial EOL-1 version was then elaborated and it permitted many experiments to be carried out. On this basis an improved EOL-2 version has been developed which is the subject of the present paper.

It should be mentioned that many symbol manipulation languages have been developed up till now. While developing EOL, use has been made of many ideas applied in the IPL-V (Newell, 1960) and COMIT (MIT, 1961) languages.

Data structure

In the EOL language the syntax is being defined not only for the program, but also for input and output data and for intermediary results such as expressions or files. Input and output data are represented in the form of strings of arbitrary characters. Up to 32 input strings can be distinguished by means of the variables

I1, I2, . . . I32

Similarly, up to 32 output strings can be distinguished by means of variables

Q1, Q2, . . . Q32

To illustrate values of strings denoted by variables I or Q we write, for example,

I1 : ABCD

Q3 : RESULT ␣ X = 1 ;

The above notation is intended exclusively for illustrative purposes, and it is neither a part of a program nor serves to determine the data. We shall make use of a similar auxiliary notation to illustrate values of expressions and files.

The input data are read character by character, so that each time the first character of the indicated string is read it is simultaneously deleted from this string. The output data are written by adding new characters at the end of the indicated output string.

There are no control characters in the input/output string. Assembling the input characters into symbols or detecting a specified character pattern is done by a program. Due to this, input and output data may have any form, the most convenient for the given problem.

Expressions

The input data are used by the program to build lists such as expressions or files. Up to 32 various expressions can be distinguished in a program. The expressions are denoted by the variables

E1, E2, . . ., E32

In order to illustrate values of expressions we write, for example,

E1 : ⁻XA°3°8^P11
E5 : ⁻TO⁻BE⁻OR⁻NOT⁻TO⁻BE
E3 : ⁻Y⁻=⁻A1⁻*⁻(⁻B⁻+⁻ALFA⁻*⁻X⁻)

Expressions are lists composed of constituents, formed of one of the marks ⁻ or ° or ^, followed by a string of characters. The marks determine the structure of the list inside the computer. They do not appear in input or output strings.

The constituents are divided into words, numbers and addresses.

A *word* consists of a string of arbitrary characters, preceded by the mark ⁻. Usually a word constitutes an independent component of the text.

A *number* is a freely chosen string of digits, possibly with an algebraic sign, and preceded by the mark °. Numbers in the EOL language permit one to perform operations on integers.

The *address* of a record is composed of a string of any characters preceded by the mark ^. These addresses serve to store positions of the records in files.

In the EOL programs there are many instructions to facilitate flexible processing of expressions, for example:

Moving a determined number of constituents from the beginning of one expression to the beginning or to the end of another expression, the original order being kept or reversed. In a special case, every variable E may play the role of the so-called stack. This property significantly facilitates the forming of recursive procedures.

Compressing several words into one word or splitting one word into separate characters.

Testing whether the initial or the final expression constituent equals a fixed word or number, or belongs to a specified class of constituents, for instance whether it is a letter.

Searching for a given word or for a specified string of words in an expression.

In the EOL programs the majority of operations handling a small amount of information are performed by means of expression processing.

Files

Up to 32 various files can be distinguished by means of variables

P1, P2, . . ., P32

Each of the files represents a list composed of any number of *records*.

In order to illustrate the value of a file, we write, for example,

$$P3 : a^{\vee}b^{\vee}c^{\vee}\sigma^{\vee}\mu^{\vee}d^{\vee}\uparrow{}^{\vee}e^{\vee}f^{\vee}g$$

where small Greek letters denote formerly defined expressions.

Every record in a file consists of a mark $^{\vee}$ followed by an *expression* or by one of the following symbols:

- σ a symbol denoting the beginning or the end of a section in a file;
- μ the sorting symbol. This symbol written in the file causes a sorting operation on a file segment between this symbol and the nearest symbol σ or the end of the file;
- $\uparrow$ a pointer which serves to distinguish one place in the file. In every file there is exactly one such pointer. Reading of a file always concerns the record which follows directly after the pointer. Writing in the file consists in writing a record in the file directly before or after the pointer.

In the EOL language there are many instructions which permit one to process files, for example:

Writing into a file an indicated expression as a new record.

Reading a record from a file and placing it at the beginning or at the end of the indicated expression.

Shifting the pointer forward until it comes across the record that satisfies a specified condition.

Resetting the pointer to the beginning of the file.

Storing the address of the record, placed directly after the pointer, as the initial constituent of an indicated expression.

Placing the pointer before the record, the address of which is the initial constituent of an indicated expression.

The last two operations permit one to manipulate addresses of records by means of instructions designed to process expressions. This enables one to determine freely the file structure, this facility being very important in many applications. For example, the EOL language often permits one to form files having a structure which allows an effective searching of information written in this file.

The files are designed to store a large amount of information; for example, to form tables for translation from one artificial language to another.

Logical variable H

The EOL programs may use a simple logical variable. It can take one of the two values "+" or "—". The value of the variable H can be changed as a result of certain instructions.

Programs

The programs written in the EOL language consist of instructions and also declarations determining switches and procedures. There is also a possibility of program modification before its execution.

Instructions

Instructions written in the EOL language cause the execution of some simple operations, such as changing of list values or interrupting a normal sequence of program execution. The structure of these instructions is very simple and resembles some computer instructions.

Every instruction consists of a key word like READ, MOVE or SEARCH followed by not more than three instruction arguments. These arguments often consist of a letter with an index e.g. A12, B1, I3. The letter specifies more exactly the method of execution of the instruction. The index determines the string expression or file to be operated on.

Several examples of instructions with a simplified explanation of their meaning are given below.

READ I3, B5, LD

Read from the string I3 successive characters, up to the first letter or digit. Form one word of the characters read, and place it at the beginning of the expression E5. Moreover, if all characters in I3 are exhausted set the variable H to "—". Characters read from I3 are removed from this string.

For example if

$$I3 : \text{␣}(\text{␣}X + 1)$$

$$E5 : {}^{-}=-Z$$

then after having executed this instruction we obtain

$$I3 : X + 1)$$

$$E5 : {}^{-}\text{␣}(\text{␣}{}^{-}=-Z$$

In the above instruction the first argument indicates the index of the input string, in this case, number 3.

The second argument determines the index of the expression and the way in which a new constituent is being added to this expression. If, instead of B5, there were Z5, the word which is read would have been added at the end of E5.

The third argument indicates the first character which is not read. In this argument LD is the symbol of a letter or a digit, similarly B is the symbol of space, and R of the remaining characters. If the third argument, instead of LD, had been B, all characters, up to the first space, would have been read from I3.

If the third instruction argument instead of LD were number 5, the instruction would cause the reading of the five initial characters from I3. If it were the literal 'A', all initial characters up to letter A would have been read.

MOVE A1, A8, '**'

Read the initial words from E1, up to $^{-}$**, the latter not being read, and place them in the reverse order at the beginning of E8. Words read from E1 are simultaneously removed from this expression. In case the word $^{-}$** is missing in E1, the entire content of this expression is moved to E8, and the variable H takes the value "—".

For example, if

E1 : $^{-}$A1^{-}+$^{-}$B^{-}**$^{-}$2

E8 : $^{-}$=$^{-}$X

then after the execution of this instruction we obtain

E1 : $^{-}$**$^{-}$2

E8 : $^{-}$B^{-}+$^{-}$A1^{-}=$^{-}$X

The first argument of the instruction indicates the way of reading words from the expression E1. If this argument were B1, instead of A1, it would mean the reading of successive initial words out of E1 without their removal.

The second instruction argument indicates that words read out of E1 are added to E8 in the reverse order, i.e. on the principle of a stack. If instead of A8 there were B8, the initial order of these words would remain unchanged.

The third argument may include, instead of '**', any literal or symbol of the type given in the former example of the instruction READ. The lack of this argument signifies moving the entire expression E1 to E8, the value of the variable H being unchanged.

SEARCH P3, B9

Shift the pointer in file P3 to the right until it is placed before the record, the first constituent of which equals the first constituent of E9. In case of reaching the record $^{v}\sigma$ or if there are no more records in the file, stop moving the pointer and give the variable H the value "—".

For example, let us assume that

E9 : $^{-}$ALFA^{-}BETA

P3 : $^{v}a^{v}\uparrow{}^{v}b^{v}c^{v}d^{v}e$

where

d : $^{-}$ALFA°3^P1

but the initial words of records ^{v}b and ^{v}c are different from $^{-}$ALFA. Then the execution of the given instruction causes

P3 : $^{v}a^{v}b^{v}c^{v}\uparrow{}^{v}d^{v}e$

Thus the pointer in P is shifted before the record d.

In the instruction, the first argument indicates the index of the file, and the second one determines the record before which the pointer is to be stopped. If the second argument were T9 instead of B9, it would specify the search in P3 for the record, the initial constituent of which is equal to any constituent comprised in E9. Similarly, the second argument equalling E9 would specify the search for the record, the initial constituents of which would be seriatim equal to all corresponding constituents of the expression E9.

GET D1, Z3

Read the expression directly following the pointer in the file P1 and put it at the end of the expression E3.

If there is no record following the pointer, set the variable H to "—".

ADD A1, 2

Add 2 to the initial constituent in E1 treated as a number.

If, for example,

E1 : °—8^{-}3

then, after executing this instruction we obtain:

E1 : °—6^{-}3

If the second argument were B3, the instruction would specify the addition of the initial number in E3 to the initial number in E1.

EQ B11, 'ALFA'

If the initial word in E11 is equal to $^{-}$ALFA, the value of the variable H is not changed, otherwise the variable H is set to "—".

GOMI L1

If the value of H equals "—", go to the instruction denoted by the label L1, and change the value of H to "+".

Switches

In the switch declaration, its identifier type and the list of labels are given. The execution of a jump instruction, containing the identifier of a switch, causes a transfer to a part of the program which is denoted by one of these labels. This instruction simultaneously specifies the expression, the initial constituent of which defines uniquely this label. If the switch is of the type INDEX, this constituent should be a number that indicates the position of the label in the list. If the switch is of type NAME—this constituent should be a word, determining the name of the label.

Let us take, for example, a switch of the type NAME

KEY : NAME FOR, IF, BEGIN ;

and assume that

E3 : $^{-}$IF^{-}END

Then the instruction

CANA B3, KEY

causes a jump to the procedure denoted by the label IF.

Switches of the type NAME permit, for example, the transfer to one of many procedures depending on the name read as a string of input characters and placed as the initial word of one of the expressions.

Procedures

Instructions and switches can be used to define procedures executing more complex operations. Let us consider, for example, a procedure, the task of which is to read a subsequent element of an arithmetic formula, and add this element as a separate constituent at the end of the expression E3. The formula is written in the form of a string of characters denoted by the variable I1, and either an arithmetic operator or an arbitrary string composed of letters and digits is thought of as the formula element. Blanks preceding an arbitrary element of the formula should be removed. For example, let

I1 : A + ␣ 3 + BETA

E3 : ⁻X1⁻=

The considered procedure being executed four times, one obtains

I1 : BETA

E3 : ⁻X1⁻=⁻A⁻+⁻3⁻+

Let us denote this procedure by the identifier ELEMENT. It can have the following form:

```
ELEMENT : PROC
  CLEAR I1, LDR        /*CLEAR INITIAL
                         BLANKS*/
  EQ I1, LD            /*IS THE INITIAL CHAR-
                         ACTER ALPHA-
                         NUMERIC*/
  GOPL L1              /*YES, GO TO L1*/
  READ I1, Z3, 1       /*READ ONE
                         CHARACTER*/
  RETURN
L1 : READ I1, Z3, RB   /*READ STRING OF
                         LETTERS AND DIGITS*/
  RETURN
END
```

The execution of the above procedure is caused by the instruction CALL ELEMENT.

After the execution of the procedure ELEMENT, the instruction that follows directly the instruction CALL is executed.

The structure of procedures in the EOL language is similar to that of the PL/1 language, which permits a block nesting and an independent definition and compilation of external procedures forming one program.

Program modification

Programs, and thus all EOL procedures written in their final form, are always related to variables I, Q, E or P, with fixed indices of a narrow range from 1 to 32. Operations on these indices or their modification during the program execution are not included in the EOL language. Therefore, substituting parameters into procedures while executing the program is not possible. This enables a significant simplification of EOL implementation. On the other hand, these limitations create some difficulties while writing programs. For instance, a small number of variables requires a careful selection of their indices. This can be inconvenient in bigger programs where, for instance, some variables should be common for several procedures.

In order to lessen these inconveniences a method has been adopted for automatic program modification *before* its performance. This method is based on the well-known idea of macro-definitions which are then included in the program by means of macro-instructions. The procedure can be initially written in the source program as macro-definitions, in which the chosen indices are written as formal parameters in the form of identifiers. These procedures can be included in the object program by means of macro-instructions, which assign actual numerical values to these identifiers. Due to this, fixed values of indices can be established by the programmer at the moment of assembling several procedures into one program. None the less, this task is not performed automatically and it still remains the programmer's duty.

Let us take the following example. The formerly given example of a procedure concerns variables I1 and E3 with fixed indices 1 and 3. In order to replace these numbers by the identifiers N and K we can use the following definition:

```
ELEMENT : DEF N, K
  CLEAR I'N, LDR
  EQ I'N, LD
  GOPL L1
  READ I'N, Z'K, 1
  RETURN
L1 : READ I'N, Z'K, RB
  RETURN
END
```

In order to include the above defined procedure with indices 1 and 3 instead of N and K in the object program, the following macro-instruction is needed:

ELEMENT : IN'PROC 1,3

If the procedure defined as a macro-definition is to be performed twice and if each time it concerns variables with different indices, two macro-instructions must be used with different values of actual parameters. As a result, this procedure will appear twice in the program, each time with a different set of indices.

However, such a case occurs in practice rather infrequently. On the other hand, the calling of a procedure is always very simple as it does not require parameter transferring during the program execution.

Summing up, the accepted method of program modification permits us to write the procedures with symbolic indices. While writing the final program the programmer is charged with some additional operations which, in practice, are not too burdensome. At these costs, the EOL language has been kept significantly simpler and the efficiency of its operation increased as compared with languages in which the writing of programs is easier.

Implementation

The EOL-2 language is being implemented in Polish ZAM computers which are binary with fixed 24-bit words. Their general organization is similar to that of many other computers, for instance, the British I.C.T. 1900 series computers.

Storage allocation

Expressions and files are held in the store in the form of lists. The list structure is different in every case.

Expressions are held in the store as lists composed of pairs of computer words. One word of such a pair comprises a part of the expression, the other word comprises the address which points to the next pair of words. A separate "free storage list" is the list to which the disused pairs of words are automatically added. If any pair of words is needed to write new information, it can be automatically taken from that list. In such a way, the allocation of storage for the expressions proceeds dynamically and automatically, and provides the maximum convenience for the programmer. That is the reason why this system has already been applied to many list-processing languages.

A file is stored so that every record is composed of two parts. The first part consists of one word which comprises the address pointing to the next record in the list. The second part constitutes the content of the record, placed in the subsequent storage words. This system is less flexible than the one accepted for expressions, as it does not automatically release the storage words occupied by disused records. However, it requires a significantly smaller number of storage words for storing records and makes easier the searching for records satisfying specified conditions.

Instructions

Every EOL instruction is represented in the ZAM computer as a 24-bit word. These words are read one after another by the program called Interpreter, which, depending on their content, causes the execution of one of the EOL instructions.

EOL language extension

EOL is a universal but relatively concise language, implemented with a basic set of instructions for symbol manipulation. More complex operations can be defined as procedures. However, in many applications such procedures can be fairly complex and their execution time rather long. In order to make them more efficient, the EOL procedures can also be written in machine language. As a rule they are based on the data structure accepted in EOL, and use the Interpreter subroutines. Thus, these procedures may be treated as EOL language extensions defining specialized operations, adapted to a given problem.

The EOL procedures written in machine language are often first written and debugged in EOL language and then translated into machine language by a programmer who is familiar with the Interpreter and the EOL data structure. This system permits one to obtain a full program documentation in EOL language and possibly to perform it in another computer, equipped with the EOL Interpreter.

An example of EOL extension for special purposes has been the experimental translation of the SAKO language to the ZAM computer machine language. The SAKO language is similar to the Manchester Autocode or FORTRAN II and it is nowadays widely applied in Poland on ZAM computers of the old type. The SAKO system was elaborated about five years ago in the Institute for Mathematical Machines in Warsaw.

The main results of the above experiment are the following. The EOL language proved its greatest usefulness during the initial but most complicated steps of translation, for example:

The analysis of the syntactic correctness of the program statements.

Transformation of arithmetic formulae into simple parentheses-free form.

Creating tables of identifiers.

After the source program has been transformed into a relatively simple form, further translation steps are easier to program in the machine language which includes the possibility of macro-definitions.

The program of the initial translation steps written in the EOL language is much more concise than an equivalent program written in machine language. However, an Interpreter which takes about 4,000 storage words must be held alongside the EOL program. Therefore, the application of the EOL language is most advantageous while translating complex languages.

The speed of operation of the translator part, written in EOL language, is initially not very great. However, after replacing some parts of this program by program segments written in the machine language, the translation speed significantly approaches that of a program written by means of a classical method.

Finally the conclusion may be drawn that the use of the EOL language can greatly simplify and speed up the writing of translators for procedure-oriented languages. The translator documentation becomes essentially simplified, which considerably facilitates the cooperation of many people on this development and maintenance. These results are the more significant the more complex is the translator.

Conclusions

Experience gained in the EOL language proves the possibility of developing a simple symbol manipulation language that is relatively efficient and easy to implement. These results have been made possible by the following features of this language:

Arbitrary form of input and output strings.

Separate device for processing short but complex expressions and for files containing a great amount of information.

Simple format of instructions and the possibility of defining more complex operations by means of procedures, which can be written also in machine language.

Possibility of automatic program modification before its performance.

The experiments we have conducted so far in EOL language application indicate its usefulness, especially for translating from one automatic programming language to another.

Acknowledgement

The author would like to express sincere thanks to all persons who concerned themselves with this project, especially to Mrs. Zdzislawa Wrotek and Mr. Jan Walasek.

References

ALLEN NEWELL (Ed.) (1960). *Information Processing Language—V Manual*, Prentice Hall, Englewood Cliffs, N.J., Second Edition.

M.I.T. (1961). *COMIT Programmers Reference Manual*, MIT Press, Cambridge, Mass.

Book Review

The Method of Summary Representation for Numerical Solution of Problems of Mathematical Physics, by G. N. Polozhii, 1965; 279 pages. (Oxford: *Pergamon Press*, 63s.)

This book describes a method of obtaining a numerical solution of partial differential equations in two or three dimensions. The method appears to be particularly suited to equations with constant coefficients to be solved over rectangular regions. The basis of the method is to replace the derivatives by finite differences, to obtain an analytical general solution of the difference equations, and then to find the arbitrary constants in the general solution so that the boundary conditions are satisfied. To obtain the solution over a rectangle the amount of arithmetic increases only as the number of points round the boundary, and not with the total number of points throughout the region. The solution obtained is an analytical one which need not be evaluated at all the mesh points.

In Chapter 1 there is the theory of the solution of one-dimensional difference equations, of order 2 and of order 4, with constant coefficients, and the relationship of the solution with the latent roots and vectors of certain tri-diagonal matrices. The treatment of this topic could have been shortened with advantage. The method proposed for obtaining the latent vectors of a tri-diagonal matrix, after the latent roots are obtained, is to use the analytical formula for the solution of the first $(n-1)$ homogeneous equations. Under some circumstances this method may give very serious errors in the latent vector due to small errors in the latent root.

In Chapter 2 the two and three-dimensional problems are considered. The general solution of the normal finite difference equations corresponding to Poisson's equation over a rectangle is obtained. From this it is shown how to obtain the particular solution over a rectangle, or a rectangle with a protuberance on one edge, or a region composed of a number of rectangles when the values of the function, or a combination of the function and its normal derivative, are given on the boundary. Other types of equations considered are the biharmonic equation in two dimensions, the parabolic heat diffusion equation in two and three dimensions, and hyperbolic equations in two or three dimensions. Finally in Chapter 2 there is a short section indicating how the method may be generalized to equations with variable coefficients.

In a supplement to the English edition there are some results obtained since the original monograph was published in 1962. An iterative method of solving the equations arising when the solution is required over a region composed of several rectangles is described first; then the method of summary representation, as the mesh size $\to 0$, is shown to be related to the method of integral transforms. Finally some particular problems, including filtration under pressure, bending of beams and plates, are considered.

This book can be recommended to those mathematicians and scientists concerned with the solution of partial differential equations. The method is not well known in this country and may be very valuable under some circumstances. The translation from Russian is first class. No printing errors, apart from those mentioned in the errata list, were noticed.

V. E. PRICE