PROGRAMMING AND COMPUTER SOFTWARE

A translation of *Programmirovanie*

Volume 7, Number 3 May-June, 1981

CONTENTS

The Russian press date (podpisano k pechati) of this issue was 5/14/1981. Publication therefore did not occur prior to this date, but must be assumed to have taken place reasonably soon thereafter.

A TEXT EDITOR WITH AN EXTENSIVE DFSTRUCTION SYSTEM FOR OS/ES

N. N. Bezrukov

UDC 51:681.3.06

The article describes the editor NEATED with its powerful and flexible instruction system. The instructions are implemented using symmetric lists. Structurally, the editor is designed as an interpreter for a single-pass programming language: the parser controls a lexic analysis block and semantic subprograms.

Text editors are among the programmers' most important tools. The availability of particular editors may significantly influence programmer productivity. The text editors currently available under OS/ES [1-4] are characterized by a relatively primitove instruction system which is not conducive to highly productive editing. The main feature of the editor NEATED described in this paper is its powerful and flexible instruction system which substantially simplifies the performance of typical text editing operations. NEATED is available in two versions: NEATED1 used interactively from the operator console, and NEATED2 used on ES 7066 display terminals. An important feature of both versions is that interactive and batch processing can be combined in a single editing sessions.

A Review of the Instruction System. In functional terms, there is not much to distinguish between a text editor, a line manipulating language (such as SNOBOL), and a macro generator (e.g., a PL/I preprocessor). In some cases, the three facilities overlap. As an example, consider the substitution of some identifier for another identifier in a program. Yet the characteristic constructs of line manipulating languages and macro generators virtually do not occur in text editors [5]. On the other hand, the instruction system of a text editor may be regarded as a simple symbol manipulating language. We therefore tried to increase the power and the convenience of the editor instruction system by incorporating the characteristic constructs of symbol manipulating languages. We moreover attempted to make each command as short as possible, so as to reduce user fatigue in keying in the commands and avoid frequent input errors. In describing the instruction system we will use a modified BNF. In our version of the BNF, square brackets denote optional strings, and braces denote iteration with zero or more repeats.

The commands may be keyed in one to a line, or in groups separated by the symbol "/". The editor displays the current line (on the console) or page (on the screen) after executing all the commands in a single line. The user can correct the displayed page with the aid of the display control keys before keying in the next command. All the NEATED commands have the same syntax:

```
<command> ::= [<range>]<command code> [<range>][<modifier>]
```

We will see in what follows that this structure allows an extensive default system, markedly simplifying the commands in many typical cases and creating significant benefits to the user. In terms if the range of each command, we usually distinguish between context editors and line editors. Context editors make it possible to define the range of a particular command by indicating the text elements that delineate the range. Line ends are entirely ignored or are considered as "eol" utility symbols. In this sense, context editors are closer to macrogenerators. Line editors, on the other hand, execute their commands using line numbers, which are either logical (e.g., the numbers in columns 73-80) or physical. They are more convenient for working with program texts, where a line is often a single operator. Both range definition methods should be regarded as complementary, and an attempt has been made in NEATED to achieve a flexible combination of the features of both context and line editors. The range of a NEATED command consists of one or several lines specified by their addresses:

```
<range> :: = <address> [, <address> | <address> [; <address>]
```

Translated from Programmirovanie, No. 3, pp. 39-48, May-June, 1981. Original article submitted July 7, 1980.

An address in NEATED is a method of locating the required line. If the range addresses are separated by a semicolon, the current line will be marked by a special pointer, the cursor (see below), before the editor proceeds to look for the line corresponding to the next address:

<address> ::= [<begin search>]{<search direction> <context>}
<begin search> ::= <logical number> | <begin text> | <end text> | <cursor>

Logical number is the number assigned to the given line of text before beginning editing, i.e., this is the physical line number in the initial text. The logical number is unrelated to the number in columns 73-80 of the given line, so that unnumbered files and raw data files with punchcard structure can be edited. Begin text (symbol "<") is the first line in the editor buffer at the current time, regardless of its logical number. Similarly, end text (symbol ">") is the last line in the current buffer. Cursor ("*") is a marker attached to the so-called current line which is printed on the console or is marked by "*" on the display screen after the completion of the last command. After the completion of each command, the current line becomes the last line corresponding to the lower bound of the right range.

Search direction is given as "+" for forward search (down the text) and "-" for backward search (up the text). The context may be defined in three alternative ways: 1) by displacement (<+5), indicating the number of lines to be skipped forward or backward from the given line; 2) by a literal which directs the search to find a line containing the given literal as a subline (* + [TEXT]). A repetition factor may precede the literal (* +3 [TEXT]), indicating how many lines containing the given string should be found; 3) by a molecule, which is defined as a part of a line between blanks. For example, the line "PUT SKIP LIST(A)"; consists of three molecules, "PUT", "SKIP", and "LIST(A);", and the corresponding search commands may be defined as <+&[PUT], or as <+&[SKIP] 2, or as <+&[LIST(A);]3.

If begin search is not specified, NEATED1 assumes by default begin text or end text, whereas NEATED2 assumes the beginning or the end of the current page on the display screen (according as forward or backward search is specified). For NEATED1 the expression <+5 thus may be written as +5. Expressions of arbitrary complexity are allowed, e.g.,

+ − 10 − 3 [PROC;] + 2 [DCL] + 3& [FIXED] 2

NEATED uses an extensive system of defaults which simplifies the composition of commands in typical cases. In particular, the following conventions apply to the addresses of left and right ranges. If the left range is undefined, the current line is assumed. If the left and the right range are defined by the same address, the line with this address is the range. If the right range is undefined, it is assumed to coincide with the left range. For example, 5A is equivalent to 5A5 or 5,5 A 5,5.

Following the general policy of OS/ES which uses English-language notation, the editor commands are represented by the first letters of the corresponding English words. The editor commands can be divided into four groups: insert–delete commands, output commands, rearrangement commands, and pseudocommands.

Insert–Delete Commands. This group includes the following commands: D (DELETE), I (INSERT before), A (ADD after), R (REPLACE), and X (EXCHANGE). For example, 3,25D deletes lines 3-25; D deletes the current line; 2I5 inserts line 5 before line 2 (deleting line 5 from its previous location); 2A5,9 adds lines 5-9 after line 2; 2,5 R7,25 is equivalent to 2,5 D /I7,25; 2,7X9,27 is equivalent to 9I2,7/8I9,27. Although all these examples use logical addressing, any other addressing method may be used, e.g.,

< + [PROC;]; * + [END;] X > − [PROC;]; * + [END;]

We see from the above examples that NEATED will fairly easily "reshuffle" the program text (this is usually a weak spot in the existing OS/ES editors). In most cases, however, new lines must be introduced into the text. This is the function of the modifier, which alters the addresses of the right range in the current command. We distinguish between input and edit modifiers.

An input modifier introduces a sequential file, a library member or a group of lines, altering the right range addresses to the addresses of the sublist representing the new text. For example, a TEXT adds the file TEXT after the current line; I (BIM) inserts the member BIM before line 5; <,> R (CAMAC) replaces the buffer content by the member CAMAC; 2,5R=/IF A THEN N = 1;/ELSE N= 0; /&replaces lines 2-5 by the corresponding expressions (the symbol "/" is the line separator, and "/&" marks end input). The input modifier not only eliminates the command INPUT available in most editors, but also creates various interesting possibilities, such as assembling a text from different components, e.g.,

A (PROLOG)/ A TEXT / A (EPILOG)

The edit modifier corrects or edits a group of lines. The right range addresses are replaced by the addresses of the new sublist, which consists of an edited text specified by the right range addresses. Since in most cases the required editing involves minimum changes in one or several lines, the updating is done using a simple language with five operations. This language largely compensates for the limited editing features of the ES 7066 display functional keys, without duplicating the functions provided by the display hardware (e.g., tabulating, substitution of single letters in a line, etc.).

The syntax of the edit modifiers is as follows:

<edit modifier> :: = [<pattern>] <operation> [<literal>].

The edit modifier specifies one of the following five operations: copy (symbol "]") substitute ("="), line add (symbol ")"), line insert (symbol "("), line delete ("¬").

1. Copy replicates several lines from one location in the text to another. For example, 2A5, 9] adds a copy of lines 5-9 after line 2. This operation requires neither pattern nor literal.

2. Substitute replaces a whole line or a part of a line by a given literal. If no pattern is specified, the entire line is replaced; if the pattern is a literal, the first string matched by this literal is replaced; if the pattern is a molecule number, the specified molecule is replaced. For example, 5R = [END;] substitutes "END;" for line 5; A[PUT] = [WRITE] inserts after the current line its copy with string "WRITE" substituted for string "PUT"; "WRITE"; R&4 = [SKIP] replaces the current line by a line in which "SKIP" has been substituted for the fourth molecule.

If no literal is specified, the sublist defined by the modifier = consists of one blank line. For example, 5A = inserts a blank line after line 5. If an integer is specified after the symbol =, an integer number of blank lines will be inserted, e.g., 7A =4 adds 4 blank lines after line 7. This feature easily "spaces out" the text on the display screen.

3. Line Insert inserts a given literal in the specified position in the line. If no pattern is specified, the literal is inserted at the beginning of the line; if the pattern is a literal, the given literal is inserted before the first occurrence of the pattern in the given line; if the pattern is a molecule number, the given literal is inserted before the corresponding molecule. For example, 9R ([LOOP:] inserts the label "LOOP:" at the beginning of line 9; R[EDIT] ([SKIP] inserts the string "SKIP" before the first occurrence of the string "EDIT" in the current line; R&5 ([END;] inserts "END;" before the fifth molecule.

4. Line Add is similar to line insert. For example, 7R) [;] adds ";" at the end of line 7; R[GET])[SKIP] adds the string SKIP after the string GET.

5. Line Delete makes it possible to delete a specified part of the line. The entire part of the line before ("(") or after (")") a specified string can be deleted. Thus, 7R [SKIP]¬ deletes the string "SKIP" from line 7; R&2¬ deletes the second molecule from the current line; 7R [STATIC]) deletes the tail of line 7 after "STATIC"; 9R [THEN](deletes the head of line 9 before "THEN."

Some of the operations may be combined. For example, 7R[THEN] (([IFM = 0] substitutes "IF M = 0" for the head of line 7; 9R [THEN])) [PUTSKIP;] substitutes "PUT SKIP;" for the tail of line 9; if the right range is defined by two different addresses, the modifier is applied in succession to all the lines in the range, i.e., an implied loop is executed. Those lines to which the modifier is inapplicable are copied unchanged. For example, 7 A 9, 22 [FIELD] = [TEXT] inserts after line 7 a copy of lines 9-22 in which the first occurrence of the string "FIELD" has been replaced by "TEXT."

Output Commands. The output commands include W (WRITE) and P (PRINT). In distinction from the other commands, the default left range for the output commands is the entire text buffer.

The command W outputs the part of the buffer specified by the left range as a sequential file or a library member. For example, W TEXT outputs the buffer as file TEXT; 5, 57W (TEXT) outputs lines 5-57 as library member TEXT.

The command P outputs the left range to printer. The printed lines are numbered on the left margin, as well as in columns 73-80, to facilitate visual search for a line with a given number. For example, the command P prints out the entire edited text; 2,79P prints out lines 2-79.

NEATED has several features for manipulating the library index. No special commands are provided for these functions: D can be used to delete a member from the library index, and R can be used to rename a member. Thus (TEXT)D deletes the member TEXT from the index; (BIM)R(TEXT) renames BIM as TEXT.

Rearrangement Commands. This group includes the commands C (CONCATENATE), S (SPLIT line into two), and B (BLANKJUSTIFY).

The command C adds a line or a group of lines specified by the right range at the end of the line specified by the first address. Leading and trailing blanks in the added lines are eliminated. For example, 2C4,5 adds at the end of line 2 the contents of lines 4 and 5. The command C can be used with an edit modifier. For example, 2C5 2⌉ adds at the end of line 2 the contents of line 5 after deleting the second molecule.

The command S splits the line specified by the first address into two lines at the given pattern boundary. The pattern boundary is specified by an edit modifier. Only two modifiers are allowed: <pattern> "(" indicates split at the left end of the pattern; <pattern> ")" specifies split at the right end of the pattern. For example, 6 S [THEN] splits line 6 into two, and the second line starts with the string "THEN"; 6 S [THEN]) splits line 6 into two, and the first line ends with the string "THEN."

The command B equalizes or shifts the lines in the left range by changing the number of leading blanks. The line content does not change. The command has several versions, e.g., 2,5B=4 uses four leading blanks in lines 2-5; 2, 5B-4 reduces the number of leading blanks by 4; 2,5B7 equalizes the number of leading blanks in lines 2-5 with that in line 7; 2,52B=5¯1 will set the number of leading blanks in lines 2-52 to five, except in those lines with one leading blank.

Pseudocommands. Pseudocommands are used to control NEATED in the process of text editing. We will only consider the most important pseudocommands in this group.

The commands U (UNFIXORDERS, meaning "clear command field") and F (FIXORDERS, meaning "keep command field") control the overwriting of executed commands on the display screen. After the command U is executed, the editor functions in a mode deleting the performed commands from the screen (except when they contain an error). Similarly, after the command F is executed, the editor switches to a mode keeping the executed commands on the screen, these commands then can be executed repeatedly by pressing the INPUT key of the display terminal. In this mode, it is easy to scan the text stepping backward or forward (roll mode) and also to update executed commands using the functional keys of the display terminals when a similar sequence of commands must be repeated several times.

The command G (GO) makes it possible to organize simple loops. The number of loop repetitions may be specified explicitly or implicitly. By specifying a command of the form G = <integer>, we execute all the preceding commands <integer> number of times. If the command specifies one or two left arguments, the commands will be repeated until one of these arguments reaches the end of the text or the cursor value exceeds the value of the second argument. For example, <+ [TEXT] R [TEXT] = [FIELD]/G replaces all the occurrences of string "TEXT" by string "FIELD"; 20 + [TEXT] R [TEXT] = [FIELD]/60G replaces all the occurrences of string "TEXT" by string "FIELD" in lines 20-60; <+ [TEXT] R [TEXT] = [FIELD]/G = 5 replaces the first five occurrences of string "TEXT" by string "FIELD."

The commands ⋕⋕, ¤¤, @@ store the specified string in the variable ⋕, ¤, @, respectively. NEATED has four line variables, whose identifiers are fixed and cannot be altered by the user: these are ", ⋕, ¤, @. These variables may be used in all commands to replace literals. The system variable " represents the last literal. The values of the variables ⋕, ¤, @ may be assigned by the corresponding pseudocommands. For example, ⋕⋕=[TEXT] assigns the value "TEXT" to the variable ⋕; "TEXT"; ¤¤= " stores the last literal in the variable ¤; 5@ stores line 5 in the variable @.

The command % (procedure definition) extends the standard instruction system by means of user defined commands. Procedures are defined with the aid of the procedure brackets "%" <letter> and "letter" "%", where letter is the procedure name. Any sequence of symbols may be enclosed between procedure brackets, which in particular may include calls to other procedures. For example, when editing programs whose text is distributed between several library members, it is often necessary to use commands of the form W/<, > D/A (<member name>). For brevity, we can define the procedure %R W/<,> D/A R%. Then %R (TEXT) is equivalent to the commands W/<,> D/Λ (TEXT). The variables ⧺, ¤, @ also may be used as procedure parameters. Two system defined procedures are available: these are "." and ":". The procedure "." is equivalent to "*+1", and the procedure ":" is equivalent to "*-1". For example, :, .D is equivalent to *-1, *+1D.

The command E (EXECUTE) dynamically switches the editor to batch mode. For example, E COM switches the editor to read commands from file COM. When end file is reached or an error is detected in one of the commands, the interactive mode is resumed.

The command Q (QUIT) terminates the NEATED session and transfers control back to OS/ES. Q does not output any data.

An important feature of NEATED is the availability of so-called <u>bicommands</u>. A <u>bicommand</u> is a command defined by two letters, instead of the usual one. Each bicommand is equivalent to two successively specified commands, and the right range of the first command is used as the left range of the second command. For example, 7RT24,55 is equivalent to 7R24,55/24,55T; 7AP (TEXT) is equivalent to 7A (TEXT)/7+1,*P; 2, 8TD is equivalent to 2,8T/2,8D; WQ (TEXT) is equivalent to W (TEXT)/Q.

Implementation Features. A basic problem in the design of text editors is the choice of the representation method for the edited text. The representation method essentially influences the instruction system, since editor commands which require for their implementation various operations that are difficult to perform in the particular text representation generally remain unimplemented or contain implementation errors. Conversely, if a useful command is easily implemented in a given representation method, it will eventually be included in the instruction system. In this sense, every text representation method as if suggests certain commands which are simply unthinkable of in other representations.

The most common representation method for the edited text is in the form of a sequential file [4, (Chap. 7), 6, 7]. The instructions which are easily realized with the sequential file representation are restricted to "delete," "replace," and insert." The range of each command is usually specified by line numbers in the edited text. Only forward context search is possible and it is generally used in order to move the cursor to a certain line. During interactive editing, the changes must be made in strict sequential order (as no backward stepping is allowed), which creates certain inconvenience. On the other hand, such editors are compact (16-20K), simple, and reliable.

In another common representation method, the edited text is stored in the direct memory as an array of fixed-length lines [3, 8]. This is also a fairly simple method, but the storage space required is proportional to the number of lines in the edited text. With this method, all the previous commands are easily implemented, plus interchange of whole lines in the edited text. However, most text manipulations involve moving the "text tail" (spacing out when new lines are inserted or compressing when lines are deleted). With large texts, this creates an excessive load on the central processor unit (CPU) when several interactive terminals engage in edit sessions at the same time. Since the method is highly wasteful of direct storage, it is often combined with the previous method: a fixed number of lines ("page") is read into memory, which corresponds to the size of the display screen or the length of a physical block on disk. In this case, a multilevel instruction system is used, with only some of the commands available at each level. To reduce the number of errors due to calling other level commands, the allowed commands are usually displayed on the screen in the form of a "menu." The STO editor developed at the Moscow Soviet Scientific-Research and Applied Institute of Computerized Management Systems is a representative of this class.

Significant reduction of memory requirements can be attained by compressing all trailing and leading blanks and representing the text as an array of variable length lines in the direct memory [8]. When the text is stored as a sequential file with fixed-length records, each line generally contains 40 (PL/I) to 60 (Assembler) trailing blanks, which may be omitted when the text is loaded into direct memory. This representation allows bidirectional context search, but it is not suitable for implementing such commands as "move a group of lines," "interchange a group of lines," copy a group of lines to a new location," etc. Nevertheless, this is a highly successful method of text representation, especially if the texts are also stored in external files in compressed form, i.e., without trailing blanks, and the lines are separated by a special "end-of-line" separator.

The most flexible instruction system can be obtained when the text is stored as a list. The list elements may be either lines of text or pointers to lines stored in a direct access file [9]. The latter method uses essentially less direct storage space, but context search involves excessively frequent accesses to disk. With list organization of the edited text, most of the line moves between different locations can be implemented by simply altering the corresponding pointers [10]. Backward context search can be simplified by introducing an additional pointer, i.e., organizing the text as a symmetric list [11]. This method was actually used in designing NEATED. Although modern programming technology recommends restricting the module length to about a hundred lines, the symmetric list representation does not essentially restrict the text size. When each line is converted into a list element, trailing and leading blanks may be eliminated, thus reducing the storage space by a factor of 3-5 for PL programs and 5-10 for Assembler programs.

With the exception of modules intended for working with libraries and the display, NEATED is written in PL/I. The list is therefore organized as a based structure allocated by the operator ALLOCATE:

```
01 CHUNK/*LIST ELEMENT*/BASED (POINT),
02 NEXT/*POINTER TO NEXT ELEMENT*/POINTER,
02 PREV/*POINTER TO PREVIOUS ELEMENT*/POINTER,
02 NUMBER/*LOGICAL LINE NUMBER*/BIN FIXED,
02 BLANKS/*NUMBER OF LEADING BLANKS*/BIN FIXED,
02 LENTEXT/*LENGTH OF LINE TEXT*/BIN FIXED,
02 TEXT/*LINE TEXT*/CHAR (LINE LEN REFER (LENTEXT));
```

This structure makes it possible to implement all the commands using a common methodology: if a line is moved, only the pointers of the corresponding list elements are changed; if a line is altered, the old element is destroyed by the operator FREE, and then a new element is created, which is linked by a pointer to the preceding and succeeding list elements; if a line is inserted, the corresponding list element is provided with a pointer linking it to the appropriate location in the list.

Storing the leading blanks in the field BLANKS not only reduces the storage space but also simplifies the implementation of rearrangement commands. For example, the command B is implemented by scanning the corresponding part of the list and updating the BLANKS field of each element.

Parsing is done by direct scanning of the symbols delivered from the lexic analyzer, without resorting to a stack. The parsed command is converted into intermediate form for subsequent interpretation. The command is interpreted only after the parsing has been successfully completed, which in some cases prevents spoiling the text by an erroneous command. Diagnostic messages for syntax errors in most cases are constructed automatically.

The lexic analyzer uses the algorithm described in [12] and has two entry points: SCAN which gets the next lexeme, and OMIT which gets the next lexeme with type checking. The allowed types are coded symbolically (e.g., the code X corresponds to literal, code A to identifier, code 9 to integer, etc.). The allowed types of the next lexeme are combined into a line, which is passed as a parameter when OMIT is called. For example, if the next lexeme in the line "5+=" is "+", CALL OMIT ('AX9'); will deliver the following automatic diagnostic message:

```
"+" FOLLOWED BY "=" INSTEAD OF IDENTIFIER, LITERAL, OR INTEGER
```

The notions of "procedure" and "macro" are virtually identical in the interpreted language. As a result, the procedure body (or the macro value) can be inserted at the lexic analysis stage. Variables are also treated as system defined macros, and the lexic analyzer substitutes for them a literal with the appropriate value.

The design experience with NEATED shows that the organization of a text editor as an interpreter and especially the introduction of a lexic analyzer are more than justified by resulting simplification of implementation and debugging. The lexic analyzer, is clearly superior to the collection of subprograms ("separate integer," "skip blanks," etc.) recommended in [13].

The communication with display terminals is organized by reading the entire screen into direct memory, with subsequent screen renew. In this way all the lines on the screen can be analyzed, while changing the list elements corresponding to the lines modified by the functional display keys.

This communication mode does not restrict the system to a particular display model (e.g., ES 7920 displays can be easily used). Changing the display only requires an appropriate display linkage module which should perform the above functions and update the load module. A shortcoming of this method is that it requires a separate direct memory buffer of corresponding screen size.

The editor has been in operation since August 1979, and it has proved to be much faster in actual use than the standard OS/ES editors. Although NEATED has a fairly extensive instruction system, the commands are easily learned by most users in about one month. Moreover, there is no need for most users to learn all the commands: a subset of the instruction system may prove quite sufficient (e.g., addressing by logical line numbers and insert–delete commands without correction modifiers). Other commands can be acquired when and if necessary.

A shortcoming of NEATED is the relatively large space requirement (about 80K), which is about twice the space needed by simple editors. However, this shortcoming is not very acute in megabyte memory computers. At the same time, user response indicates that NEATED will cope with almost any text manipulation task (although sometimes not without difficulty). NEATED has been applied to edit FORTRAN IV and PL/I programs.

NEATED is available as part of the NEAT software package [14-16] developed at the Ukrainian Scientific-Research Institute of Psychology.

LITERATURE CITED

1. OS/ES. Utilities. Data Manipulation, Printer Output, and Punching. Programming Manual [in Russian], Moscow (1977).
2. OS/ES. Time Sharing Mode. Command Language. Programming Manual [in Russian], Moscow (1978).
3. A. G. Rubin and V. K. Smirnov, "An interactive editor for alphanumeric displays under OS/ES," Preprint Inst. Prikl. Mat. Akad. Nauk SSSR, No. 117, Moscow (1976).
4. I. V. Vel'bitskii, V. N. Khodakovskii, and L. I. Sholmov, A Programming Technology Package for ES and BÉSM-6 Computers [in Russian], Statistika, Moscow (1980).
5. A. Dam and D. E. Rice, "On-line text editing: a survey," Comput. Surveys, 3, No. 3, 93 (Sept. 1971).
6. S. R. Bourne, "A design for a text editor," Software – Pract. Exper., 1, No. 1, 73 (1970).
7. P. Hazel, "A general-purpose text editor for OS/360," Software – Pract. Exper., 4, No. 3, 389 (1974).
8. P. Deutsch and B. W. Lampson, "An on-line editor," CACM, 10, No. 12, 793 (1967).
9. I. A. Macleod, "Design and implementation of a display oriented text editor," Software – Pract. Exper., 7, No. 4, 771 (1977).
10. D. Foster, List Processing [Russian translation], Mir, Moscow (1974).
11. J. Weizenbaum, "Symmetric list processor," CACM, 6, No. 9, 524 (1963).
12. N. N. Bezrukov, "Compilation principles for the RYaOD language," in: System and Theoretical Programming [in Russian], Inst. Kibern. Akad. Nauk Ukr. SSR, Kiev (1979), p. 39.
13. B. W. Kernighan and P. J. Plauger, Software Tools, Addison-Wesley, Reading, Mass. (1976).
14. N. N. Bezrukov, "NEATPL – a tool for simplifying PL/1 program debugging," Programmirovanie, No. 5, 87 (1978).
15. N. N. Bezrukov, "Generation of a program for printing the document header from the description of the header structure," Programmirovanie, No. 6, 92 (1979).
16. N. N. Bezrukov, "A modification of the Floyd–Evans language," Programmirovanie, No. 4, 53 (1979).