Some Results of Research on Automatic Programming in Eastern Europe

WŁADYSŁAW TURSKI

Computation Center
Polish Academy of Sciences, Warsaw, Poland

Introductory Notes

The author of the present article considers it impossible to cover in a single paper all interesting aspects of the vast area of research on automatic programming in Eastern Europe. Therefore the reader will kindly accept the author's apologies for inadequate description, omission of many interesting points, and, frequently, too brief a discussion of papers mentioned. It is hoped that the reader will consider the present article as a kind of summary rather than a complete survey.

As for references that may be made to the relevant literature in English the author suggests that Professor Carr's report on a visit to the Soviet Union [*5*] should be consulted for general impressions;[1] one of the sections of Professor Chorafas's book [*6*] provides an excellent though short analysis of the main trends in Soviet research, and Capla's paper [*3*] presents many interesting data on hardware. It is however our duty to warn the reader that all three references are based on somewhat obsolete information.

The reference list appended to the present article is necessarily incomplete, though it includes a few more items than are referred to in the text. For the papers having English translations, corresponding references are given; however, no systematic search has been conducted in order to find all translations available. Two Western [*4*, *7*] and one Soviet [*68*] reference papers are listed and any of them may be consulted for missing papers. Also, a recent paper [*19a*] should be noted, which is pertinent to this discussion.

There was no attempt either to make comparison with relevant Western research or to find out mutual influences, except when a problem was of obviously international character, e.g., ALGOL compilers.

Finally, if in relating somebody else's work the author has committed blunders or overlooked misprints, he wishes to take the blame for such oversights.

All opinions expressed are the author's only, and do not necessarily agree with those of his colleagues or superiors.

1. Soviet Union

1.1 Fundamentals of Programming

On considering Soviet achievements in automatic programming one can easily notice that a great part of the work done on the problem is connected in one way or another with Professor Lyapunov of Moscow University. The main results of the research carried out under Lyapunov's guidance and influence up to 1953 were published for the first time in the first volume of *Problemy Kibernetiki* (Problems of Cybernetics), where papers by Lyapunov, his collaborators, and his pupils predominate, [*27*, *41*, *42*, *46*, *55*, *72*].

It seems that apart from several very interesting exceptions, discussed in Section 1.5, all major work on automatic programming in

[1]See also, Contemporary Soviet Computers, *Datamation* **9**, No. 11, 24–30 (1963).

the USSR is done in notation, and uses concepts and definitions conceived by Lyapunov's team. Since, moreover, Lyapunov's paper [*41*] provides a very convenient departure point for our sight-seeing tour of Soviet developments in our problem, we shall begin our venture with a brief discussion of that remarkable paper, leaving for later pages the analysis of the different approaches to the ever-exciting question of how to cause a soulless machine to perform the ingenious task of converting mathematical thoughts into sequences of binary codes.

Computational procedures involved in solving various problems on digital computers may generally be considered as algorithms for information processing. The initial data constitute the information to be processed; the results constitute the processed information.

Such information-processing algorithms consist of more or less uniform stages which are executed by the computer one after another in some ordered sequence. Each stage itself may be considered to perform some strictly defined information processing. We shall say that each of these stages is performed with the help of an *operator*. Consecutive performance of operators will be called *product of operators*. It often happens that the order of execution of some operators depends on the results of performance of other operators; thus there arises the necessity of having operations that check the fulfillment (or otherwise) of *logical conditions* which govern the order of execution of operators. These conditions are frequently represented by *predicates*.[1a]

Finally, a third set of elements is formed by so-called *steering parameters* which are used to indicate repetitive use of some operators. The number of repetitions depends on the value of these parameters.

All algorithms are to be built up of these three kinds of elements: operators, logical conditions, and steering parameters. This is achieved by means of *calculational schemes*, i.e., products of operators and logical conditions.

Each logical condition, which may be simply a predicate, is followed by an arrow indicating transfer of control in case the condition is not fulfilled. If the condition is fulfilled, the next (in the left-to-right sense) operator is being executed. In Lyapunov's notation arrows are broken in two parts: one, pointing up, occupies the place immediately after

[1a]This expression is in a sense very troublesome; the corresponding Russian term *предикат* has been translated into English by other authors as: logical variable [*12*] or logic-algebra function [*71*]. It is hoped that the adopted term, *predicate*, will cause no ambiguity; in addition, the following remark may be useful: a predicate is a function of some logical or arithmetical relation, defined in such a way that, if the relation is true, the predicate assumes value one, otherwise the value is zero. Predicates are closely related to *if clauses* in ALGOL.

the logical condition to which it belongs, the other part, pointing down, precedes the operator, or another logical condition as the case may be, which is to be obeyed, or checked, if the original condition happens to be not satisfied. Arrows are identified by numbers written above the principal line occupied by the calculational scheme. This may be observed in the example given in Appendix 1.

In the calculational schemes (which are independent of any hardware) operators are represented by capital letters with subscripts indicating dependence on steering parameters. A product of operators may be recorded in the condensed form:

$$A_1 \cdot A_2 \cdot A_3 \cdot \cdots \cdot A_n = \prod_{i=1}^{n} A_i.$$

Logical conditions are represented by small letters, and predicates take the form of functions whose arguments are the conditions to be checked. Predicates may be of one of the following four types:

(i) $p(|a| \geqslant |b|)$,
(ii) $p(a \leqslant b)$,
(iii) $p(a = b)$,
(iv) $p(a \neq b)$,

where a and b are variables whose values should be known before evaluation of the predicate is initiated.

Lyapunov frequently uses a rule which says that, if non-fulfillment of a condition leads to skipping of just one operator, arrows are omitted altogether in the calculational scheme. Arrows without identifying numbers may be used with the understanding that transfer of control is made from an unnumbered up-arrow to the first following down-arrow. In order to illustrate this notation we shall consider two examples.

Example 1. Let the operator A_k generate and print out integer k^2. The following three schemes are equivalent and represent the calculational scheme for printing (a) squares of even numbers if p_s is a logical condition which is satisfied when s is even, or (b) squares of odd numbers if p_s is a logical condition which is satisfied when s is odd:

$$p_1 \uparrow A_1 \downarrow p_2 \uparrow A_2 \downarrow \cdots p_n \uparrow A_n = \prod_{i=1}^{n} p_i \uparrow A_i \downarrow = \prod_{i=1}^{n} p_i A_i. \qquad (1.1.1)$$

Example 2. Let us consider a calculational scheme for solving simultaneous linear equations:

$$\sum_{i=1}^{n} a_{ji} x_i = a_{j,n+1}, \qquad j = 1, 2, \ldots, n. \qquad (1.1.2)$$

Assuming that the diagonal elements of the matrix are not zeros we may solve the set (1.1.2) by the following scheme:

$$\left(\prod_{i=1}^{n}\prod_{j=1}^{n} B_{ij}\, p(i=j)\, C \prod_{k=1}^{n+1} A_{ijk}\right) \prod_{m=1}^{n} D_m \qquad (1.1.3)$$

where operator B_{ij} generates $c = a_{ji}/a_{ii}$, operator A_{ijk} generates $a'_{jk} = a_{jk} - ca_{ik}$ and transfers it into the location previously occupied by a_{jk}, operator D_m generates $x_m = a_{m,n+1}/a_{mm}$, operator C replaces c by zero, and parentheses have their customary mathematical meaning.

Since the digital computers possess finite memories it is essential to make the computational schemes as short as possible. This aim is achieved by executing similar operators, i.e., operators differing by the value of steering parameters only, by the same pieces of code. That is to say, we would like to record in the machine memory only the initial form of the operators forming the scheme, and make the machine not only execute them but also prepare re-execution of some of the operators. For this purpose Lyapunov introduces several types of *control operators*. Their role consists in preparing the machine memory for execution of consecutive operators and necessary control transfers. The following list is, in Lyapunov's opinion, complete enough to secure solution of quite complicated problems:

(1) readdressing operators,
(2) restoring operators,
(3) transfer operators,
(4) forming operators,
(5) parameter change operators,
(6) parameter introduce operators,
(7) operators for switching logical conditions.

For some types of programs it will be essential to introduce freely *additional logical conditions* which will secure the desired sequencing of operators.

Lyapunov insists on separation of two similar terms: *computational scheme* and *programming scheme*. In his terminology a computational scheme deals with abstract operators; a programming scheme deals with the programs (or pieces of code) which realize these operators. Thus his definition of programming scheme may be formulated as follows. A programming scheme for a given problem is a product of program realizations of operators and logical conditions (that is, a realization of their product) which possesses the following property: When all these realizations of operators (and logical conditions) are fed into a computer together with the needed initial data, automatic processing of recorded information will be initiated and will not be stopped until the desired solution is found. At the beginning of the work of each operator, the machine memory will be in a condition which makes the performance of the operator possible.

This somewhat diffused definition may be explained by the rule for obtaining programming schemes from calculational ones. The rule reads as follows: To obtain the programming scheme one should furnish the corresponding calculational scheme with (i) control operators which will provide such conditions of machine memory as are necessary for execution of successive calculational operators and (ii) logical conditions which will secure desired sequencing of calculational operators.

Now we shall consider the purposes of control operators in more detail.

Readdressing operator is a generic name given to pieces of code which change the address parts of some of the program instructions, viz., in those instructions that are relevant to parameter-dependent operators. The changes thus introduced are prescribed by changing values of parameters.

Readdressing operators are denoted by capital **F** and the following convention is obeyed: $\mathbf{F}(i)$ increases the value of the parameter i by 1, while $\mathbf{F}(ki)$ or $\mathbf{F}^k(i)$ increases the value of the parameter i by k. A notation like $\mathbf{F}(3i, 5j)$, equivalent to $\mathbf{F}(3i) \cdot \mathbf{F}(5j)$, is sometimes used in order to point out that it may be possible to change both parameters by the same instruction.

Restoring operators form a subclass of readdressing operators singled out for their ability to restore the initial value of a parameter. Lyapunov does not introduce any special symbols for the restoring operators.

Transfer operators transfer numerical data from one location to another; in symbolic notation they assume the form $[a \rightarrow b]$, where a denotes data to be transferred and b the location to which this data is to be transferred. (Using often the same symbol to represent both contents and location, Lyapunov intentionally smooths over the difference between the two.)

Forming operators is the name given to pieces of code which generate the initial form of some operators of the program. These operators transfer previously set instructions, or combinations of them, into prescribed locations in the body of the program, thus forming a new operator from separate pieces.

A forming operator may sometimes be used instead of a restoring one, this being especially convenient when the number of readdressings to be performed on a parameter is not known in advance.

Generally, if the forming operator generates the operator B, the notation $\mathbf{\Phi}(B)$ is used to denote this fact; when, however, the forming operator is used as a restoring operator, a symbolism similar to transfer operators is employed; e.g., if the forming operator is to restore the initial value s of the parameter i we shall write $\{s \rightarrow i\}$.

It may be of some interest to note that, while restoring operators are likely to perpetuate minor mutilations of a program already recorded, forming operators are more likely to avoid this fault.

Parameter change operators are generalized readdressing operators, the generalization consisting in application rather than in formal structure, viz., parameter change operators are used when not only readdressing controlled by a parameter is desired but also introduction of the numerical value of altered parameter into arithmetic formulas. Notation is the same in both cases.

Sometimes it happens that the value of a parameter becomes known as a result of execution of some operators. Then, a necessity arises for having special *parameter introduce operators* which would introduce the value of a parameter thus obtained into operators which depend on it. The forming operators quite often may be considered as introducing initial values of parameters.

In the preparation of programs for fairly complicated problems, a situation may occur where logical conditions governing control transfers may depend on results of previously performed calculations; e.g., depending on results obtained, we may need to check either the condition α or the condition β. Moreover, it may happen that not only the choice of conditions, but also eventual transfer of control may depend on previously obtained results. In such cases the *operators for switching logical conditions* are used. Those operators are most frequently realized by means of transfer of previously stored pieces of code which may be readdressed or changed in any other way if necessary.

In Appendix 1 the interested reader will find an example of a rather complex programming scheme.

Let us now consider the so-called standard Lyapunov operators [*27*]:

A, arithmetical (computational) operator;
P, logical operator (predicate or logical condition);
F, readdressing operator;
Φ, forming operator;
O, restoring operator.

All other operators will be called nonstandard operators and denoted by **H.**

Standard operators are, in a sense, homogeneous and connected.

The homogeneity of standard operators consists in the functional similarity of all machine instructions composing the code representation of a given operator; that is to say, either all instructions perform some arithmetical operations, or all of them are readdressing instructions, etc. In other words, the task of the entire group of instructions realizing a given operator may be formulated by a single "command." The group of instructions forming the machine representation of an operator is said to be connected because it consists of a string of instructions with one "entrance" only, and all these instructions follow each other tightly, with no "empty" locations; there are no alternative paths inside of one group of instructions.

These two properties are extremely helpful not only in programming but, and especially so, in debugging procedures, when a program written in machine code is analyzed, since they make it possible to break the program into pieces corresponding to original operators. Moreover, some new types of standard operators were found on the basis of such an analysis (cf., e.g., checking procedures described in Section 1.2).

It is perhaps worthwhile to observe that Lyapunov's notation, although meant to be machine-independent, is obviously pertinent to one-address machines. This becomes particularly clear when one compares Lyapunov's notation (or its modification due to Yu. I. Yanov [*72*]) with the notation adopted in [*31*][2]. For the sake of consistency we shall henceforth abandon Lyapunov's notation and adopt Yanov's, thoroughly described in [*31, 32*]. The main difference between them lies in the symbolism employed to denote "jumps." Yanov uses small α instead of Lyapunov's small p and replaces arrows by so-called left and right strokes, $\underset{i}{\llcorner}$ and $\underset{i}{\lrcorner}$. The notation $\mathbf{A}_{i1}\alpha \underset{m}{\llcorner} \mathbf{A}_{i2} \cdots \underset{m}{\lrcorner} \mathbf{A}_{i3}$ means that, if $\alpha = 1$, control is transferred to $\mathbf{A}_{i2}$, while if $\alpha = 0$, control is transferred to $\mathbf{A}_{i3}$.

The notation used in [*27*], obviously designed for three-address computers, differs from that of Lyapunov and Yanov in the following manner. The ith logical condition (or "predicate") is followed by an open bracket $\overset{n}{\underset{m}{[}}$ with two numbers, denoting the ordinal numbers of operators to which the control is to be transferred; the upper number is the ordinal number of the operator to be executed if the preceding logical condition is satisfied, otherwise the control is transferred to the operator designated by the lower number. The mth and nth operators are to be preceded by closed half-brackets, $\underset{i}{\rfloor}$ and $\overset{i}{\rceil}$, respectively. For

[2]Notation similar to this is adopted by Ershov [*12*].

example, letting **S** represent either **A** or **P** (in the sense of Lyapunov's standard operators), $\mathbf{S}_1\overset{3}{\rceil}\mathbf{S}_2\mathbf{S}_3\overset{2}{\underset{4}{[}}\underset{3}{\rfloor}\mathbf{S}_4$ means that, if $\mathbf{S}_3$ happens to be true, the next operator to be executed is $\mathbf{S}_2$, otherwise $\mathbf{S}_4$. In order to make this notation more concise, in some cases the lower parts of open brackets and the entire corresponding closed brackets are omitted. This is always so if after the *i*th logical condition the lower number is $i+1$; e.g., the scheme just mentioned may be rewritten $\mathbf{S}_1\overset{3}{\rceil}\mathbf{S}_2\mathbf{S}_3\overset{2}{\lceil}\mathbf{S}_4$.

In Soviet literature three types of repetitive calculations are distinguished:

(1) iterative cycle $\underset{2}{\rfloor}\mathbf{A}_1\mathbf{P}_2\underset{1}{\lfloor}$,

(2) cycle with readdressing $\underset{3}{\rfloor}\mathbf{A}_1\mathbf{F}_2\mathbf{P}_3\underset{1}{\lfloor}$,

(3) cycle with readdressing and restoring $\underset{3}{\rfloor}\mathbf{A}_1\mathbf{F}_2\mathbf{P}_3\underset{1}{\lfloor}\mathbf{O}_4$.

In all these cases $\mathbf{A}_1$ denotes the computational part of a cycle; $\mathbf{F}_2$ denotes readdressing; $\mathbf{P}_2$, $\mathbf{P}_3$ are conditions checking whether the cycle is completed; and $\mathbf{O}_4$ is a restoring operator.

It is of interest to note that most Soviet-made digital computers possess built-in facilities for all three types of cycles.

Generally speaking, Lyapunov recommends a multilevel preparation of programs, viz., to begin with a programmer should split the algorithm into a few big parts and conduct a thorough study of "how these pieces work together". Then, the parts should be split into operators in order to provide a calculational scheme. The last stage of "intelligent" work consists in writing down the programming scheme. Afterwards, the "mechanical" job of coding should be performed in order to produce the machine code for the given problem. This approach to programming, so simple in principle, becomes much more complicated when one tries to optimize (in any sense) the programming scheme. Lyapunov has pointed out that there exist at least two different methods for simplifying and optimizing programming schemes. One method consists in formulating formal criteria of goodness of programming schemes and inventing formal rules of transforming schemes into equivalent but simplified ones; another method, strongly advocated by Lyapunov himself, consists in material transformations of programming schemes. The second method requires a certain amount of ingenuity on the programmer's part and rather fair knowledge of the mathematics behind the algorithm, hence this method is in a sense useless as a possible basis for automatic programming. In the paper under consideration, Lyapunov gives a number of examples which have become standard in Soviet nonautomatic programming technique.

The first method has been thoroughly studied by Yu. I. Yanov in a series of extremely interesting papers [*70*, *71*, *72*]. This study, closely related to both Lyapunov ideas of programming and A. A. Markov's theory of algorithms [*47*], is a splendid example of far-reaching research, conducted with the help of the most modern apparatus of formal logic. Unfortunately Yanov's work is outside the scope of the present survey, and thus we shall not discuss it in any detail. One remark, however, may be made without formal discussion; viz., judging from frequent quotations of the series in Soviet and foreign papers on the theory of autoprogramming this work represents not only a formal but also a very powerful practical tool for the authors of autocodes.

Before closing this section, we present one more fundamental concept due to the Soviet school of automatic programming, namely the so-called "logical scale."[3]

Let us suppose that the calculational scheme is divided into n stages each containing a predicate p assuming one of two possible values 0 or 1 depending on the ordinal number of the stage currently under execution. Let the machine memory cells be of L bits, then s consecutive cells of the memory will be called a logical scale for the program if they satisfy three conditions:

(i) $s \geqslant n/L$.

(ii) There exists a possibility of counting individual bits in the array of s cells according to the principle that bits $0, 1, \ldots, L-1$ belong to the first cell, bits $L, L+1, \ldots, 2L-1$ to the second, etc.

(iii) All bits corresponding to those stages in which $p = 1$ are filled with unities, all others being zeros.

Thus, once such a scale is recorded in machine memory, it easily may be used to govern all control transfers dependent on p. There are, generally speaking, two possible ways of using logical scales. The first one is to introduce a digital probe, i.e., a number represented by unity on the first bit location of a cell and zeros on all others. The Boolean product of the digital probe and the first cell of the logical scale is not zero if (and only if) $p = 1$ for the first stage of calculations. Hence, if a computer is provided with a built-in check for zero accumulator, organization of transfer control becomes rather trivial. For the following

[3]Unfortunately, I was not able to determine when and where this concept appeared in print for the first time, though I am told by Dr. Greniewski that this device was commonly used in Poland and Czechoslovakia as early as 1956. Soviet authors, *e.g.* [*32*], [*41*], unanimously consider M. R. Shura-Bura as being the inventor of the logical scale.

stages the unity in the digital probe is automatically shifted by one bit location to the right after accomplishment of the current stage.

The second way of using the logical scale is more efficient for computers with no built-in checks for zero accumulator, but with a check for negative (or positive, as the case may be) content of this register. Suppose that the very first bit location of the cell is the sign bit. Then, loading the accumulator with the first cell of the logical scale, we may organize the jumps in such a manner that jumps conditioned by negative accumulator content will agree with what is desired in the case of $p = 1$. As soon as the first stage is accomplished the entire scale is shifted by one bit "leftward" and so on.

The logical scale method has numerous applications in the practice of programming and is by no means limited to the checking of logical conditions. It may be used successfully, e.g., for identification, on a list of n items differing by one characteristic; we form a logical scale sufficiently long to have n bits at least, and for items having the given property we enter unities on the corresponding bits of the scale. Such an application of logical scales is employed by Ershov in his algorithm for translation of arithmetical expressions (cf. Section 1.3).

Bibliographical notes. Lyapunov's theory, outlined above, may be conveniently studied in a book by Kitov and Krinitskiĭ [*32*], which is meant as a textbook for university students. Part of an earlier book by Kitov [*31*] has been translated into English and appeared in Vol. II of "Frontier Research on Digital Computers," edited by J. W. Carr, III, University of North Carolina. A refined formal work on programming schemes for algorithms is presented in Ershov's paper [*13*]. A very fine example of material transformations of programming schemes is due to N. G. Arsentieva [*1*] who considered generally applicable algorithms of linear algebra and simplifications that may be made when, say, dealing with symmetric or diagonal matrices. Iliffe [*24*] has used the Lyapunov/Yanov notation and results of Yanov's research on formal transformations of programming schemes. In an interesting paper by R. I. Podlovchenko [*51*] some formal methods for transforming programs (not necessarily algorithms) are discussed.

1.2 Development of "Programs Which Program"[4]

A first attempt to automatize the programming procedures was made in 1954 by two scientific workers of the Computing Laboratory of the

[4]In this article, in analogy to Eastern European usage, the terms "programs which program" or "programming programs," or the abbreviation PP, are used for what English-speaking authors have variously called translators, processors, compilers, etc.

Soviet Academy of Sciences, V. M. Kurochkin and L. N. Korolev. The main results of their work were two programs for the BESM computer which performed translation of arithmetical formulas and assembling of programs according to their programming schemes.[5]

At the same time in the Mathematical Institute of SAS the PP-1 translator for logical, readdressing, and restoring operators was constructed by Miss G. S. Bagrinovskaya, E. Z. Lyubimskiĭ, and S. S. Kamynin. A revised and enlarged version of this translator, the PP-2, was built by S. S. Kamynin, E. Z. Lyubimskiĭ, M. R. Shura-Bura, Miss E. S. Lukhovitskaya, V. S. Shtarkman, and I. B. Żadykhajlo. This programming program for the Strela computer is the first Soviet fully automatic translator which produces an object program in machine code from a source program written in the form of a programming scheme (with some additional information added). Arithmetic formulas in the source program are given explicitly. However, there is one thing to remember. Up to the present time, no known Soviet-made computer possesses an alphanumerical input device. All computers can accept numerically coded information only.[6] Thus, a source program written in accordance with the rules of a program that programs (e.g., PP-2) should be numerically coded before the actual key punching.

Such a procedure has one considerable advantage; viz., Soviet scientists, when composing rules for an external language, are not restricted by the limited number of different symbols available on keyboards. As a matter of fact, introducing a new symbol into the set of allowed ones means just one thing, that one more numerical code is to be added to the already existing coding dictionary. This advantage is really a big one and makes programming in external languages very easy, but the price paid for it is not small either. A large number of coders are employed for performing a very tedious and laborious task—translating alphabetic expressions and conventional mathematical symbols into their "numerical representation." This work is not only dull, it is also apt to cause many hardly detectable errors; to prevent errors, each program is coded by two different persons [*27*] and results are checked against each other.

Now we proceed to the main principles of PP-2 as a typical programming program.

The first stage of programming consists in writing down the program-

[5]The historical part of this section is based on Ershov's lecture [*12*].

[6]The first known exception to this rule is the Soviet-made Ural-2, installed in the Computation Centre of the Polish Academy of Sciences, with paper tape readers attached to it.

ming scheme (cf. Section 1.1). Next, each **A** operator should be specified, by supplying (i) mathematical formulas describing calculational procedures incorporated in it, and (ii) a list of quantities which are used by these formulas. At this stage a distinction is to be made between *results* and *intermediary results*. Every number generated in the course of execution of an operator **A**, which serves as an operand for subsequent pieces of that operator but is not its output, and therefore irrelevant for other operators, is called an intermediary result. Cells allocated for these quantities are called *working cells* or *temporary storages*. It is quite irrelevant from a general point of view which cells are used as working ones.

For all **P** operators, corresponding logical functions should be specified together with necessary logical conditions, relations serving as arguments for predicates, and directories for transfer of control. Specifications for **F** operators include list of operators to be readdressed and number of corresponding parameters. For **Φ** operators, specified items are quantities to be loaded in standard cells and ordinal numbers of operators in which quantities should be replaced by contents of standard cells. For **O** operators, the ordinal numbers of the operators to be restored should be given. Finally, for nonstandard operators, **H**, pieces of code should be recorded.

The next stage of programming consists in the preparation of the so-called list of *conventional numbers* which serve, in a sense, as identifiers. Each conventional number consists of 12 bits divided in two groups. The first consisting of three or four bits, represents the type of the quantity associated with the given number: variable, operator, working cell, etc. The second identifies the individual quantity within an array of quantities which are similar, i.e., have an identical first group. This method is rather inconvenient, since it does not provide facilities for indexing. In some programming programs, a third group of bits is introduced just for that purpose.

Arithmetical formulas are written according to the following rules:

(1) A formula may contain arithmetical operations which are built into the computer, or which are executed by standard subroutines permanently recorded in machine memory. This is not a very severe restriction since the set of allowed operations is fairly extensive.

(2) There are no priority rules for operations, hence a suitable number of pairs of parentheses should be introduced.

(3) All formulas should be linearized (in the typographical sense).

(4) The quantity which is the result of the formula is written to the right of the "=" sign.

Thus, e.g., the formula

$$z = \frac{1 + a + b \cdot \ln x - \sqrt{(c + a + 1) \cdot \ln x} + d \cdot e^{(1+a+b \cdot \ln x)}}{a + b \cdot \ln x} \qquad (1.2.1)$$

should be rewritten as

$$(1 + a + (b \cdot \ln x) - \sqrt{((c + a + 1) \cdot \ln x)} + (d \cdot \exp(1 + a + (b \cdot \ln x)))) \,/\, (a + (b \cdot \ln x)) = z. \qquad (1.2.2)$$

As we have already said, variable identifiers as well as operation symbols are replaced by conventional numbers manually.

The algorithm for producing the object code from arithmetic formulas may be described as follows [*42*]:

(i) A left-to-right scan finds the first operation that can be executed, a corresponding piece of code using conventional numbers is produced in such a way that the result is loaded into the first empty working cell. The coded part of the formula is erased and in its place the number of the working cell (address) is recorded.

(ii) The entire formula is scanned to the end, and all similar operations with identical operands are replaced by the number of this working cell. At this stage one restriction is compulsory, viz., not really all similar operations are replaced but only those that follow a left-hand (open) parenthesis. This restriction is somewhat milder in the case of add and multiply operations; namely, operations similar to that already coded are replaced when they follow not only the open parenthesis but also the "+" and "·" signs.

(iii) Trivial parentheses, i.e., parentheses embracing one quantity only, are erased.

(iv) Control is transferred to the first stage, unless the formula is already reduced to one of the two forms $r_k = x$ or $a = x$, where r_k denotes the kth working cell. In that case, the final piece of code, the "final load," is produced.

The algorithm described above will produce the following code for formula (1.2.2):

$$\begin{array}{ll} 1 + a = r_1 & r_4 - r_7 = r_8 \\ \ln x = r_2 & \exp r_4 = r_9 \\ b \cdot r_2 = r_3 & d \cdot r_9 = r_{10} \\ r_1 + r_3 = r_4 & r_8 + r_{10} = r_{11} \\ c + r_1 = r_5 & a + r_3 = r_{12} \\ r_5 \cdot r_2 = r_6 & r_{11}/r_{12} = z. \\ \sqrt{r_6} = r_7 & \end{array} \qquad (1.2.3)$$

Shtarkman proposed an interesting economy algorithm [*55*] for reducing the number of working cells required. This algorithm is discussed in some detail in Section 1.3.

Logical conditions in PP-2 are coded by means of Lukhovitskaya's algorithm [*46*]. This algorithm decodes any logical statement built up of predicates (cf. Section 1.1) and the logical operators $\vee$ (alternation), (conjunction), and $-$ (negation), and produces a code which secures execution of all prescribed control transfers. Let us assume that P is either 1 or 0, and control is to be transferred to A if $P = 1$, and to B otherwise. Lukhovitskaya's algorithm works as follows.

(i) The first pair of corresponding left and right parentheses is found.

(ii) Partial outlets A' and B' for that pair are determined:

(a) The first "·" following the right parenthesis of the pair is spotted. If between the parenthesis and the "·" sign any "$\vee$" occurs we search for "·" following the next closed parenthesis. In either case, A' is the first predicate, p_i, following the conjunction so determined. If, as may happen, there is no conjunction sign in the remaining part of the formula, the partial outlet A' is identical with A.

(b) To determine B' we search for the first "$\vee$" sign following the right parenthesis. B' is the first p_i following this sign. As in (a), if no $\vee$ is found, B' is identical with B.

(iii) If the content of the parenthesis just considered had been negated, we change A' into B', and *vice versa*.

(iv) To determine control transfers inside of a parentheses pair we apply the following procedures:

(a) We load some fixed cells with A' and B'.

(b) In a right-to-left scan we find the first predicate p_i (on the first scan this is simply the last one inside the parentheses pair) and produce the code: **go to if** $p_i = 1$ **then** A' **else** B'.

(c) For all the following (in the right-to-left sense) predicates we produce code equivalent to **if** $p_i = 0$ **then go to** B' **else go to** preceding (in the given sense!) predicate. This procedure is interrupted as soon as the first $\vee$ is found.

(d) Having discovered the $\vee$ sign, B' is replaced by the predicate occupying the position next to the $\vee$ sign (from the right).

(e) We go back to (b) unless the open (left) parenthesis is reached.

(v) The entire parentheses pair and its content is erased and replaced by the first predicate, then we go back to (i).

The programming program PP-2 includes, besides the two algorithms for arithmetical and logical operators just described, many other facilities, like the Shtarkman algorithm for economy of working cells, debugging procedures which output storage addresses allocated to conventional numbers, etc.

One more feature of this program should be mentioned. The PP-2 produces an object program on cards. Thus, in order to perform actual calculations another deck of cards has to be fed into the computer input device.

This program serves as a basis for many other programming programs. One of these is a very elaborate programming system developed at the Lomonosov University, Moscow, under the guidance of M. R. Shura-Bura and N. P. Trifonov, for the STRELA-4 computer [*64*]. This system uses many basic concepts of PP-2 (such as Shtarkman's economy algorithm, conventional numbers, and so on, but possesses some new features which deserve special attention. For instance, the system treats a program as an assembly of *standard subprograms.* Standard subprograms, sometimes called *standard subschemes,* are those parts of programming schemes which may be written independently of the program in which they are used, and are "connected" with the main program by means of some parameters (one such subprogram is, e.g., a cycle). Each of the standard subprograms is given a name, i.e., a conventional abbreviation. The programming system accepts programs with such abbreviated notation, and before actual translation is initiated, the full programming scheme of the program is constructed from the set of programming schemes of standard subprograms. This method is considered as useful for two reasons: (i) the set of standard operators acceptable for translation may be frozen; (ii) the algorithm for replacing names by programming schemes may be very simple since it does not perform the actual coding. This is especially worthwhile for it allows the introduction of new standard subprograms with extreme ease. Actual coding in the system is done by an algorithm which uses a limited and *frozen* set of standard operators.[7]

When producing codes the programming system takes care not only to economize the quantity of working cells but also to minimize the number of instructions involved in representing the given operator in the form of machine code.

A very interesting novelty of this system is the checking procedure. Several such procedures are available in the system.[8] We shall briefly

[7]The reader will note the difference between the meaning attached to the word "subprogram" in this section and the conventional understanding of the term. To avoid possible misunderstandings, a subprogram in the conventional sense will be called subroutine.

[8]The original version of checking procedures is due to T. A. Trosman of the Mathematical Institute of SAS. The working version is due to N. V. Sedova, V. V. Voevodin, Yan Fu-Tsen, and E. A. Zhogolev (cf. [*64*]).

discuss two of them. The *control program* is the program for imitating the well-known step-by-step checking which used to be performed manually. The main idea of the program is to use the object program as input information for the control program which, according to the programmer's option, accepts one or more of the object program instructions and, once more optionally, either executes these instructions one by one printing out partial results, or prints relevant control transfers, or just the instructions themselves in octal or decimal form. Moreover, a special facility is provided for introducing so-called *test data* to be processed by a desired sequence of instructions of the object program with output of the results of that process. This part of the checking procedures may be used in two modes: Either the control is returned to the checking procedure program after the execution of the chosen instruction and output of the relevant data, or the entire sequence of instructions is executed without transferring control to the checking program. The checking procedures described up to now are what one could unmistakably call the automation of step-by-step manual checking. There is, however, an additional set of checking procedures, called *program analyzer*, which is itself a most interesting example of a refined approach to programming processes. The analyzer takes a program written in machine language and processes it so as to obtain as an output the original source program written in Lyapunov's operator language. Of course, two conditions must be satisfied to secure the successful use of the analyzer:

(i) The program to be analyzed must have been produced in strict accordance with definite rules (as it is if the programming system is used to code the programming scheme in machine language).
(ii) There must be but one operator for which a given sequence of operations serves as the machine representation.

If the programming program uses any economizing algorithms then programs reproduced by this procedure will differ from the original ones in the sense that, e.g., all arithmetic formulas will be reproduced in optimal order. Jokingly speaking, original and analyzer-reproduced programs are identical *modulo* wasteful instructions. This feature of the analyzer constitutes one of its principal advantages, enabling it to be used for checking the quality of source programs. Another possible use is reconstruction of the source program from the object program when, say, the original source program is lost, or when a straightened version is desired.

The analyzer employed in the programming system is, in its author's own words, "just one of the first attempts to produce such a program" and thus undoubtedly will need a great deal of improvement before it gets its final form.

Generally speaking, the analyzer determines the type of the operator to which a given instruction belongs, by examining the operation part of the instruction. For some instructions that may be used by two or more operators, some additional information is derived from two sources. First, it is supposed that a point to which control is transferred by a logical operator is always the beginning of some operator. This is used as auxiliary information in dividing the code into parts corresponding to operators. Secondly, it is presumed that the object program consists of three blocks: instructions and relative constants, absolute constants, and data. Thus, from the value of the address part of an instruction, it may easily be inferred what type of operator this instruction belongs to.

Unfortunately no more details of the analyzer can be given without going too far into a technical description of the STRELA-4 computer and details of other components of the programming system.

We now turn to two of Ershov's programming programs: the program for the BESM computer (the PPBESM system) and that for the STRELA-3 computer (the PPS system). These systems are described in [*11*, *12*].

PPBESM is quite similar to PP-2, described earlier, and thus we shall give only those details which are different. There are two features which are, from the user's point of view, most interesting and important, though rather trivial from the programmer's side, viz,. automatic storage allocation and priority of arithmetic operations. From a formal point of view, programs are written in somewhat different manner, though still strongly influenced by Lyapunov's notation. For instance, in PPS the following notation is employed. A variable is denoted by a small letter, which may be supplied with any number of subscripts; however, the use of subscripts is limited to denoting dependence on cycle parameters. Cycles themselves are represented by curled brackets enclosing the cycle. To open a cycle the symbol $\left\{\begin{matrix} i_K \\ i=i_H \end{matrix}\right.$ is used, i_H denoting the initial value of subscript i, and i_K its final value. If $i_H=0$ it may be omitted, thus $\left\{\begin{matrix} 10 \\ s \end{matrix}\right.$ means that the operations between this symbol and the corresponding $\}$ are to be executed repeatedly for $s = 0,1,2, \ldots, 10$. The only admissible step for the cycle parameter is 1. Instead of giving the explicit value i_K, a relation of the form $i \leqslant m$ may be employed as an upper limit for the cycle parameter.

Besides arithmetic operators, represented by formulas, five other types of operators are allowed:

logical operator	**Λ**,
restoring operator	**B**,
nonstandard operator	**H**,
readdressing operator	**Π**,
repeat and check operator	**DC**.

Each operator is represented by its symbol followed by a set of parentheses inside of which information about the operator is given. Operator symbols may be subscripted; this time, however, subscripts denote labels of the operators. There is plenty of freedom in writing arithmetic formulas. As a matter of fact, only two common symbols are not allowed, namely, the fractional bar (horizontal) and the root symbol. Both should be replaced by powers. There are two symbols for multiplication: "×" and "·"; the latter is often omitted (implied multiplication). In the following example arguments of functions are underlined in order to show the difference in the meaning of the symbols.

$$z_i \cos\underline{(a+b)(a-b)} + \ln z_j^2 + d\,\mathrm{tg}\underline{(f+a)} \times (a-b) \Rightarrow r_s \cdot \qquad (1.2.4)$$

Logical operators are written in the form

$$\mathbf{\Lambda}\left(a \;\square\; b \underset{N_1}{\overset{N_2}{[}}\right) \qquad (1.2.5)$$

where $a \,\square\, b$ stands for any one of the relations $a<b$, $a\leqslant b$, $|a|<|b|$, $|a|\leqslant|b|$, $a=b$, i.e., the same set of relations as used by Lyapunov (cf. Section 1.1). The meaning of the logical operators is similar to the predicates in previously discussed papers, $\mathbf{\Lambda}\,(a<b\underset{N_1}{\overset{N_2}{[}})$ meaning "**go to if** $a<b$ **then** N_2 **else** N_1", where N_1 and N_2 are subscripts of operator symbols. Unconditional jumps are represented by $\underset{N}{\overset{N}{[}}$, and conditional jumps depending on the content of the test register are written as $\underset{N_2}{\overset{N_1}{[}}$ with the rule: If the content of the test register is 1, then transfer the control to the operator labeled (subscripted) N_1, otherwise jump to N_2.

A few words should be said about the **DC** operator. Any part of the source program may be included between two **DC** operators. This will result in the following procedure being executed. After actual calculations according to this part of the program have been performed, the content of the main storage is summed up and the sum thus obtained is recorded. Then the relevant calculations are repeated and a second summation is accomplished. If the two sums agree, the entire content of the main storage is recorded on magnetic tape. If the sums disagree, the computation process is interrupted and the operator (human) either calls the repair man or, by repeating the calculations for a third time, tries to choose between the two earlier results.

Much attention has been given to economizing the translation time for PPS. A very fine example of an economy algorithm is discussed in Section 1.3.

The PPS is some 1200 three-address instructions long, and translates source programs at the speed of about 35 three-address instructions of the object program per minute [*12*].

Bibliographical notes. There are at least three complete descriptions of different programming programs used in the Soviet Union. The PP-2 is described in Volume 1 of *Problemy Kibernetiki*; the programming system for Strela-4 and the programming program for BESM are published as separate books [*11*] and [*61*]. Besides, many more algorithms are scattered throughout different issues of *PK* and other Soviet journals.

A formalized algorithm for translation of arithmetic formulas similar to that of Lyubimskiĭ is given by Evžen Kindler [*30*], see Appendix 2.

1.3 Some Results of Optimization of Translators

Addressing the participants of the Symposium on Mechanization of Thought Processes, A. P. Ershov said [*12*] that there are two paths for the future development of programming programs (translators) which are of equal importance. One way leads to more flexible translators allowing for the use of more complicated forms of source programs, thus reducing the time that the programmer must spend in preparation of the programs. The alternate way leads to translators which would produce object programs more quickly and, what is perhaps even more important, would produce object programs that work more quickly and use fewer working cells.

It is certainly possible that the technological development which has recently been progressing at a tremendous pace may still accelerate, thus providing us with more and more powerful computers; but nevertheless *ceteris paribus* it will always be desirable to possess translators which are more nearly optimal in both senses quoted by Ershov.

In the present section we shall not follow the pattern of the two preceding ones. We shall present neither a historical account, nor a full discussion of pertinent work accomplished in the Soviet Union. Nor shall we attempt to exhaust the subject. We will select just two algorithms (very similar in a sense) and look at them more closely.

To begin with, we will study the *Shtarkman algorithm* for reducing the number of working cells required as temporary storage for intermediary results in the evaluation of arithmetic formulas [*55*]. The number of working cells required by the program is equal to the greatest number of working cells required for any single operator (in Lyapunov's sense). As far as the Shtarkman algorithm is concerned, it is essential that the computer for which the algorithm was constructed be a three-address machine. The general structure of the instruction address part for the STRELA-4 is IA, IIA, IIIA, where IA and IIA denote addresses of operands, and IIIA denotes the address of the cell into which the result is loaded. (For some operations not all of these are used.) Let us consider the address parts of a piece of code corresponding to an operator (see Table I).

TABLE I

Distribution of conventional numbers in the address parts of the instructions corresponding to an operator[a]

IA	IIA	IIIA	Regions of existence		
		r_1	r_1		
	r_1				
r_1		r_2			r_2
r_2					
	r_2	r_3	r_3		
r_3		r_4		r_4	
r_4	r_2				
	r_4	r_5	r_5		
r_5	r_4				

[a] r_i denotes the conventional number of the ith intermediary result.

Of course, the simplest allocation rule would be to assign individual working cells to all conventional numbers $r_1, r_2, \ldots, r_5$. However, a glance at Table I reveals that this would be a poor rule, since, e.g., the same working cell may be used for keeping results r_1 and, say, r_5. As a matter of fact, we need only two working cells to serve the example of Table I. The set of instructions during whose execution the working cell loaded with r_j is tied up, is called the *region of existence of the intermediary result* r_j. This set begins with the instruction generating r_j and ends with the instruction that precedes the last one in which r_j is used as an operand. (Regions of existence of intermediary results in Table I are shown in the last three columns.)

Thus, the following rule may be formulated: Intermediary results whose regions of existence do not overlap may be assigned the same working cell. According to this rule, the conventional numbers in the example should be replaced by a reduced set of conventional numbers, i.e., a set in which different numbers are attached to intermediary results whose regions of existence do not intersect. In other words, r_1, r_2, and r_5 should be replaced by r'_1, and r_3 and r_4 by r'_2. This example explains the main principles of the algorithm.

With reference to the example considered in the previous section [cf. formulas (1.2.3)] it may be interesting to compare the old code with one obtained by application of the Shtarkman algorithm:

$$\begin{array}{ll} 1 + a = r_4 & r_2 - r_3 = r_3 \\ \ln x = r_3 & \exp r_3 = r_2 \\ b \cdot r_3 = r_1 & d \cdot r_2 = r_2 \\ r_4 + r_1 = r_2 & r_3 + r_2 = r_2 \\ c + r_4 = r_4 & a + r_1 = r_1 \\ r_4 \cdot r_3 = r_3 & r_2 / r_1 = z. \\ \sqrt{r_3} = r_3 & \end{array} \tag{1.3.1}$$

Instead of 12 working cells only four are used.

The Shtarkman algorithm is a typical one-pass algorithm which works as follows: All instructions forming an **A** or an **F** operator, i.e., the only ones that may possibly use working cells, are scanned starting with the last one. During this procedure a special table T is produced in which all the conventional numbers representing the intermediary results are listed, providing that the operation currently looked at belongs to the region of existence of the pertinent intermediary result. The table T is produced according to the following rules. If IIIA contains a conventional number of an intermediary result, the relevant row in T is cleared. If a conventional number occurs in either IA or IIA, and in T there is no row containing that number, the first empty row of T is loaded with that number. If there is no empty row in T, a new row is added. Now, the table T possesses the following properties:

(i) Each row of T contains conventional numbers of intermediary results whose regions of existence do not overlap.

(ii) The conventional numbers of any two intermediary results with overlapping regions of existence are to be found in different rows of T.

(iii) The number of rows of T starts at zero at the beginning of the procedure and grows to the minimum number of working cells required at the end.

The unique assignment of working cells to rows of T and replacement of the r_i's occuring in a row by the corresponding working cell address solves the economy problem.

Shtarkman gives the following programming scheme for the algorithm just described. In the scheme we have preserved Shtarkman's original notation, hoping that the reader will not be inconvenienced too much:

$$\begin{gathered}\mathbf{\Pi}_1 \underset{3}{\overset{3}{[}} \overset{7}{\rceil} \mathbf{O}_2 \underset{1}{\overset{1}{\rfloor}} \textbf{Ш}_3 \overset{16}{\lceil} \mathbf{\Pi}_4 \mathbf{\Omega}_5 \underset{7}{\overset{7}{[}} \underset{8}{\rfloor} \underset{15}{\overset{15}{\rfloor}} \textbf{З}_6 \underset{5}{\overset{5}{\rfloor}} \textbf{Ш}_7 \overset{2}{\lceil} \mathbf{P}_8 \underset{6}{\lfloor} \mathbf{\Pi}_9 \underset{11}{\rfloor} \textbf{Ш}_{10} \\ \overset{14}{\lceil} \mathbf{P}_{11} \underset{10}{\lfloor} \boldsymbol{\omega}_{12} \underset{15}{\overset{15}{[}} \textbf{И}_{13} \underset{15}{\overset{15}{[}} \overset{10}{\rceil} \mathbf{B}_{14} \underset{12}{\overset{12}{\rfloor}} \underset{13}{\overset{13}{\rfloor}} \mathbf{\Phi}_{15} \underset{6}{\overset{6}{[}} \overset{3}{\rceil} \mathbf{R}_{16}.\end{gathered} \tag{1.3.2}$$

There are three special cells α, β, and γ which are to keep the following information:

α contains the currently examined operation.
β contains the currently examined fraction of the address part of the operation contained in α (IA, IIA, or IIIA).
γ contains the number of the first empty row of T.

The algorithm works in three stages for each operation (corresponding to IA, IIA, and IIIA).

Meaning of operators in Shtarkman's algorithm:

$\mathbf{\Pi}_1$	Preparatory operator.
$\mathbf{O}_2$	Transfers content of α back to operator code.
$\textbf{Ш}_3$	Picks next operation from the operator code, loads α with it, and checks for the end of the operator.
$\mathbf{\Pi}_4$	Preparatory operator for three stages necessary for each operation.
$\mathbf{\Omega}_5$	Clears operator $\boldsymbol{\omega}_{12}$ and skips next operator.
$\textbf{З}_6$	Loads $\boldsymbol{\omega}_{12}$.
$\textbf{Ш}_7$	Puts next piece of address part into β, "tunes" operator $\mathbf{\Phi}_{15}$ for loading next conventional number into prescribed address, checks for end of the operation.
$\mathbf{P}_8$	Checks whether content of β is a conventional number of any of the intermediary results.
$\mathbf{\Pi}_9$	Prepares search through T, loads γ with new address.
$\textbf{Ш}_{10}$	Picks up next row of T; if the row is empty, loads γ with its address; checks for end of T.

P_{11} Checks whether the selected row of T contains the conventional number searched for.

ω_{12} This operator, if present, causes the next one to be skipped (operator is to be cleared for the IIIA stage, and restored for IA and IIA).

$И_{13}$ Clears pertinent conventional number from T.

B_{14} Loads pertinent conventional number into T.

Φ_{15} Forms the new address (erases the old conventional number in IA, IIA, or IIIA, depending on the stage currently performed, and replaces it by the new conventional number).

R_{16} Outputs the number of working cells required for the examined operator code.

The second algorithm we are about to discuss here is a part of a more general formal algorithm due to Ershov [*10*]. Since, however, the remaining parts of the general algorithm are rather well known, and since the English speaking reader may consult the translation of the original paper, we shall give here only the principles of the "subalgorithm" used to reduce the number of instructions involved in the object program representation of an arithmetic formula belonging to a source program.

Ershov's algorithm will be presented as an ALGOL procedure. This form is comparatively easy to follow, and was used by the author of the present article in a course of lectures on ALGOL given in 1962. For the sake of convenience, some features of the original paper are retained, though they prevent a more concise notation.

procedure Economize code for a given operation first address (a) second address: (b) third address: (c) control bit: (sigma) operation number: (theta);

comment We suppose that the given operation is one of n operations generated by the given operator. All these n operations are stored as a two-dimensional array $L[1{:}n, 1{:}5]$. We suspect that there may be some repetitions in the array which should be deleted by the procedure. Array $R[1{:}t, 1{:}5]$ consists of resultant operations, i.e., those which perform the assignment for the formulas forming the given operator, t is the number of formulas in the operator. Integers assembled in the array $C1[1{:}m]$ are called conventional numbers of the first kind and represent variables and constants pertinent to the operator, there being m different variables and constants. Boolean arrays $B[1{:}n]$ and $B1[1{:}m]$ are logical scales. At the first entry to the procedure (for the given operator), the preparatory part, preceding label 1, should be executed, this being controlled by a Boolean variable "first entry." Labels correspond to individual operators forming Ershov's algorithm. The seventh operator is skipped as irrelevant to the procedure. Structure of Ershov's function is explained below. On calling the procedure, sigma = 0 (always);

```
value $a$, $b$, $c$, theta, sigma; integer $a$, $b$, $c$, sigma, theta;
begin integer $i$, $j$, $p$, $s$;
    if $\neg$ first entry then go to 1;

    for $i = 1$ step 1 until $n$ do
    begin for $j = 1$ step 1 until 5 do $L[i, j] := 0$;
        $B[i] :=$ false
    end for;
    for $i = 1$ step 1 until $m$ do $B1[i] :=$ false; $p := 0$;
    1: if $c \neq 0$ then go to 9;
    2: $s :=$ Ershov function (theta, $a$, $b$);
    3: for $j = 1$ step 1 until 5 do if $L[s, j] \neq 0$ go to 6;
    4: if $a = L[s, 1] \wedge b = L[s, 2] \wedge$ theta $= L[s, 5] \wedge$ sigma $=$
       $L[s, 4]$ then go to outlet 1;
    5: if $s < n$ then $s := s + 1$ else $s := 1$; go to 3;
    6: for $i = 1$ step 1 until $m$ do
              if $C1[i] = a \vee b = C1[i]$ then
                    begin $B[s] :=$ true;
                          $B1[i] :=$ true
                    end;
    8: $L[s, 1] := a$; $L[s, 2] := b$; $L[s, 5] :=$ theta; go to outlet 1;
    9: if $B1[c]$ then for $i = 1$ step 1 until $n$ do
            if $B[i] \wedge (L[i, 1] = c \vee L[i, 2] = c \vee L[i, 3] = c)$
                then $L[i, 4] := 1$;
       $R[p, 1] := a$; $R[p, 2] := b$; $R[p, 3] := c$; $R[p, 4] :=$ sigma;
       $R[p, 5] :=$ theta; $p := p + 1$; go to outlet 2;
end of procedure
```

We come to outlet 2 if the given operation is resultative; we come to outlet 1 if the operation is not resultative. In the latter case s denotes an intermediary result. (Numerically, s is the address of an operation in the array of operations L). Note that owing to the operator labeled 9, no operation may be identical with any of the operations loaded into L *before* the current operation.

The idea behind the algorithm is not really new, but the use of the special Ershov function to determine the initial searching point in the array L is certainly original. The usual thing to do in this type of algorithm is to take, instead of $s :=$ Ershov function (theta, a, b), the relation $s := 1$. Ershov claims that by using a generator of pseudorandom numbers with almost constant distribution over the interval $[1, n]$ and studying statistically the structures of arithmetic operators, it is possible to construct functions that will be easy to evaluate and will considerably reduce the time consumed in the execution of the algorithm.[9]

Before leaving this interesting algorithm we shall point out that logical scales are used here for quite different purposes than their original application described in Section 1.1.

Bibliographical notes. Ershov's paper [*10*] contains one more algorithm for reducing the number of working cells. Interesting remarks on economy of memory requirements can be found in a paper by S. S. Lavrov [*40*], which are discussed in Section 1.6. Many useful devices for obtaining optimal codes are contained in [*64*] and [*11*]. Two recent papers [*67*] and [*48*] are devoted to problems of automatic checking and symbolic addresses used in programming programs.

1.4 Soviet Work on ALGOL

In autumn of 1958 at the Mathematical Institute of the Soviet Academy of Sciences a research program was undertaken to construct an input language for electronic computers that would satisfy two conditions:

(i) It ought to be as easy to learn as possible.
(ii) It should be machine-independent.

As a basis for such a language the Zurich version of ALGOL (ALGOL-58) was originally adopted, but then a number of generalizations were introduced, resulting in a more convenient notation. The proposed generalizations almost coincided with those changes in ALGOL which were introduced at the Paris conference; thus most of the discrepancies between the proposed input language and ALGOL were removed. It may be interesting to note that this coincidence in proposals arose in the work of two completely independent groups, and that the formal side of the proposed alterations is quite different.

In a paper by Lyubimskiĭ [*43*] some remarks are given concerning disadvantages and shortcomings of ALGOL-60; these remarks were made at a Symposium organized by the Mathematical Institute. The remarks have a rather formal character and go along lines proposed by Peter Naur in *Algol Bulletin.* Since, moreover, some of the features of ALGOL-60, considered by Lyubimskiĭ as liable to cause misunder-

[9]Algorithms of that type usually take up time proportionally to n^2; Ershov's algorithm needs time proportional to n.

standings, are considered by other authors as purposely unrestricted, we shall not give any details of Lyubimskiĭ's view.

The most advanced Soviet work on ALGOL is presented in a book by Ershov, Kozhukhin, and Voloshin [*16*]. The book defines what has been called on several occasions the Siberian dialect of ALGOL. It is certainly true that the language proposed in the book differs substantially from ALGOL, though it remains in close relation to it; as a matter of fact, the proposed language incorporates ALGOL in the sense that any correct ALGOL statement is a correct sentence of the Siberian dialect, although the converse is not true.

Compared with ALGOL-60, the Siberian language possesses many additional features; the most important of them are listed below.

(1) It is permissible to use complete arrays in arithmetical statements; thus it is possible to write matrix operations just as arithmetic expressions, without involving complex structures of procedures. This generalization, permitting a simple variable to denote a multidimensional array (and not only a scalar element of it), calls for extended possible type declarations for simple variables, for it would become necessary to declare the structure, i.e., dimensions and bounds of subscripts, for single variables. In addition, arithmetic operators and standard functions should be redefined to allow operations on arrays represented by simple variables. For convenience, the metalinguistic term "array" should be divided into "matrix" and "array", the latter being reserved for more than two-dimensional structures and for such two-dimensional arrays that do not obey the rules of matrix algebra.

Finally, two new types of statements—*forming statement* and *composing statement* for building arrays from components having fewer dimensions and smaller components of the same dimensions, respectively—are introduced. [It is curious to observe that in completely different problems similar statements are defined and used by Klouček and Vlček (cf. Section 3.1).]

(2) The Siberian language allows for use of more complex relations in Boolean expressions, viz., it is possible to use *chain relations* such as:

if $a[1] = \cdots = a[n] = 0$ **then** S;

if $A \geqslant a > b \geqslant B$ **then go to** Novosibirsk.

This extension is connected with the abbreviated form of sequences depending on parameters. Thus, in correct Siberian language it is possible to write

$T[c_1] = \cdots = T[c_2]$ instead of, say,

$T[c_1] = T[c_1 + 1] = T[c_1 + 2] = T[c_1 + 3]$ (when $c_2 = c_1 + 3$).

Generally, $T[c_1]w \cdots wT[c_2]$ denotes any sequence in which w is a delimiter, and c_1 and c_2 are the initial and final values of a parameter.

(3) In the Siberian language it is possible to enter a block at a place other than its head. This is achieved by introducing *compound labels*. A compound label takes the form $A.B$, a dot separating the components of the label. When used in **go to** statements, it signifies transfer to a block labeled A, with *inner* label B. This preserves the inherently local character of labels and makes it possible to enter the labeled statement of any block from the outside.

(4) There is a STOP statement in the Siberian language. This statement is equivalent to a dummy statement but has no successors whatsoever.

(5) There are other minor formal differences, all conceived to simplify the programmer's work.

(6) More fundamental differences are to be found in the concept of *upper indices*. Upper indices, when used, denote variables which are evaluated in some dynamical order. The current value of an upper index shows the number of the currently used value of a given variable. Consecutive values of upper-indexed variables are often obtained from recursive arithmetic formulas, e.g.,

$$x^{i+1} := a \times x^{i-2} + x^{i-1} + c \times x^{i}. \qquad (1.4.1)$$

There is obvious simplification in translation of formulas written with the help of upper indices, since only those different values of an upper-indexed variable need be kept in machine memory which are used by the formulas. For the (usual) lower-indexed variables, i.e. those arranged as an array, all values must be preserved.

(7) It is proposed that in the Siberian language a new rule for formal parameters called by value be adopted. Usually, for a given procedure, formal parameters called by value may be used as arguments of procedures only. In the Siberian language any formal parameter may be called by value, thus becoming local to the procedure body; however, if it is one of the "results" of the procedure, its actually computed value is assigned to that actual parameter which corresponds to the "resulting" formal parameter.

The concept of "call by value" in standard ALGOL-60 can be explained informally as follows: For the formal parameters entering the value list the corresponding actual parameters are evaluated once on entry into the procedure body (when the procedure is activated) and the values thus obtained are assigned to the formal parameters by means of implicit assignment statements which precede the first **begin** of the procedure body. The formal parameters called by value will in subsequent statements be treated as normal local variables of the body. Hence, if in the course of execution of the body, new values are assigned to such parameters, these new values would be inaccessible outside of the body, i.e., for instance, in the block which activated the procedure. This means that no "results" of the procedure could be treated in this way. In the Siberian language for such resultative formal parameters, called by value, implicit assignment statements are introduced, which are executed immediately after the final **end** of the procedure has been reached. These assignments act in a manner exactly opposite to the ones performed on entry into the body, i.e., the values of formal parameters obtained last (in the dynamic sense) are assigned to the corresponding actual parameters, specified in the procedure statement.

This rule, while not enriching the "expressive power" of the language, may however become very convenient by reducing the overall number of registers needed to represent variables of the program.

(8) The Siberian language preserves the ALGOL-58 concepts of using procedures and functions with empty parameter lists as actual parameters, and allows for the use of several identifiers for one procedure.

(9) In order to make it possible to use the customary forms of formulas in which specifications of certain variables are given after the formulas themselves—e.g., $x := y + a$, where $y = \sin(\pi/z)$—new declarations are introduced. These declarations are formally similar to assignment statements, as e.g., the above used $y = \sin(\pi/z)$. Like all other declarations they have no operative meaning and serve only to introduce initial values of parameters or variables. Besides, this type of declaration leads in some cases to considerable simplification and time economy in object programs.

Ershov [*16*] gives a thorough formalized description of the Siberian language. In addition there are many interesting examples of procedures and programs written in that language, together with a very useful Siberian-ALGOL dictionary and glossary of new terms.

It is hoped that this brief section will inform the reader about the deep and far-reaching research on ALGOL conducted in the USSR.

As to practical applications of this theoretical research, very little can be said now. At the present (end of 1962) there are three groups working on implementation of ALGOL. Two of them, led by Ershov (Novosibirsk) and Shura-Bura (Moscow), were stated[10] to expect to achieve their goal by early 1963[10a]. Both are trying to implement dialects

[10]Private communication.

[10a]*Added in proof.* During the Kiev conference on automatic programming (Kiev, June 1963) Shura-Bura's ALGOL compiler TA-2 was demonstrated. A thorough description of this translator and the ALGOL dialect adopted is to be found in a paper:

of ALGOL on the M-20 computer with card input. If and when this is achieved, the M-20 will become the first Soviet computer to possess alphanumerical input operated in the USSR (cf. Sections 2.1 and 2.3). The third group is led by V. M. Kurochkin [*38*].

Bibliographical notes. Besides the quoted book [*16*], there is a paper by Ershov [*14*] which gives some "shop" details about the construction of the Siberian language and future translators for it. A number of ALGOL-written algorithms has appeared in *Zhurnal Vychislit. Mat. i Matem. Fiziki.*

1.5 Non-Lyapunovian Approach

The ideas discussed in preceding sections refer to concepts of programming schemes and programming programs (PP's) of a certain, though not sharply defined, type. For all the PP's hitherto considered, it is typical to possess considerable generality of form, and virtually no distinction is made between programming of problems taken from different fields. In other words, these PP's are not dependent on the actual content of the source program.

In the pertinent Soviet literature, apart from variations of this approach, one may find two fundamentally different approaches which are discussed briefly in this section.

First of all, we discuss the autoprogramming method developed at the Computation Center of the Ukrainian Academy of Sciences headed by V. M. Glushkov [*19, 58, 59*].

This method tends to eliminate a considerable amount of highly intellectual work which must be performed by skilled programmers and educated mathematicians when Lyapunov's multilevel preparation procedure is used (cf. Section 1.1). Glushkov [*19*] rightly observes that the method proposed by Lyapunov, and subsequently adopted in many programming programs, leaves to the programmer the very difficult problem of choosing the right algorithm for the given problem and then preparing calculational schemes, whereas with little additional work this stage of programming could also be automatized.

Let us consider a finite set of typical mathematically stated problems, which will be called the *class of allowed problems*, or CAP for short. The elements of this set are not specific problems but types of problems, each consisting, generally speaking, of an infinite variety of specific

ALGOL-60 Translator, [Shura-Bura, M. R., and Lyubimskiĭ, E. Z., *Zhur. Vych. Mat. i Mat. Fiz.* 4, No. 1, 96-112, (1964)]. During the same meeting it was announced that the Kiev Institute of Cybernetics, headed by Academician V. M. Glushkov, produced another ALGOL compiler for the KIEV computer.

problems. For each element of CAP there exists a finite number of known algorithms. All these algorithms should be programmed once and for all and kept in machine memory (perhaps in some auxiliary type of storage). In addition there should be one special program for each element of CAP which will do the "thinking" on selecting an algorithm for a specific problem. Of course, there are many principles influencing the choice of an algorithm for a given problem, such as accuracy desired, individual features of the specific problem, memory requirements for each of the possibly applicable algorithms, and so on; but it is quite clear that once these principles are formalized there is a possibility of programming the choice procedure.

All partial programming programs provide control facilities of two types. One of these serves for checking the calculation procedure, the other checks whether the chosen algorithm, the machine parameters and the specific problem are compatible, in the sense that "it is possible to solve the given problem, with given accuracy, by means of the chosen algorithm, on the given computer."

Partial programming programs that perform actual programming (coding) of specific problems use standard subprograms for performing tasks that are common to more than one partial programming program, e.g., coding arithmetic statements. Glushkov suggests that some nonnumerical programs might be included in the library of subprograms. These would perform a number of easily formalized analytical transformations of data (e.g., differentiation and integration of elementary functions, substitution and reduction of polynomials, etc.). To this question we shall return in Section 1.6. Glushkov says that the input information, or source program, for a computer endowed with an autocode of the proposed type would consist of statements like: "solve the set of simultaneous differential equations

$$dx_1/dt = -5x_2^2, \qquad dx_2/dt = x_1 + \cos 2x_2,$$

with initial conditions $t = 0$, $x_1 = x_2 = 1$ over the range $0 \leqslant t \leqslant 10$, with accuracy better than 0.0001; and produce values of x_1 and x_2 for $t - 1, 2, \ldots, 10$." After reading such input the computer will "itself" choose the right algorithm and perform the calculations.

There is no known realization of such an autocode yet; however, first attempts to implement similar methods were made by Stogniĭ, who produced working programs for solving linear differential equations and formulated some criteria for choosing one of two algorithms: that of Runge-Kutta and that of Adams (Störmer) type. Unfortunately, the criteria were not formalized and thus the actual decisions have had to be made by programmers. In [*58*] Stogniĭ gives a program for reading sets of equations (coded manually letter by letter) and producing object programs for either of two algorithms. In [*59*] he proposes a scheme for differentiating (analytically) equations of the type $z = f(x, y)$, $y = y(x)$, and uses this scheme for a program that solves differential equations in the vicinity of singular points.

Glushkov's ideas may be thought of, in a sense, as generalizations of Lyapunov's principles. Now we proceed to a quite original approach advocated by Yu. A. Shreĭder [*53*] and [*54*] and his colleagues V. A. Kozmidiadi and V. S. Cherniavskiĭ [*37*]. The ideas pertinent to this approach are as yet neither embodied in any practical representation, nor even worked out in enough detail to permit such an embodiment. However, this approach opens such exciting perspectives and leads to so profound an analysis of programming that any review of relevant Soviet work not referring to these ideas would be incomplete.

The basic idea behind the proposed method of programming is to consider a computer as a language that includes, as a sublanguage, the programming language. In this connection, a *language* consists of (i) a set of material objects (symbols, electronic components) constituting an *alphabet* of the language, (ii) rules for composing *correctly formed* finite subsets (words, expressions, formulas, registers, accumulator) constituting a formalized syntax of the language, together with (iii) some interpretation of the *texts*, i.e., finite sets of correctly formed subsets. Interpretation is defined by rules constituting the *semantics* of the language and, by and large, may be understood as a mapping of the *linguistic system* (set of all correctly formed subsets) onto a set of objects called meaning or values. Thus, an electronic computer and an abacus may be considered as languages. This somewhat shocking statement becomes clear when one recalls that the abacus (used for calculations and not as, say, a kind of trolley) is useless unless some interpretation rules are given; in other words, to use the abacus, one has to interpret correctly formed combinations of beads on wires as numbers.

Similarly, an electronic computer without proper semantics is just a meaningless heap of circuits, elements, and racks. For a computer treated as a language, the alphabet consists of primitive elements like flip-flops or ferrite cores (sometimes called variables) which may assume one of two possible states (called values of variables). Such variables are interpreted as material variables, and the states of material variables are interpreted as the values zero or one.

Aggregations of these variables, e.g., registers, are correctly formed expressions interpreted as "variable words," and the states of these aggregations are interpreted as numbers or instructions. Which specific interpretation is to be attached to a given state of a given aggregation depends on, e.g., the state of another aggregation.

Sequences of expressions, corresponding to processes executed by the machine, may well be considered as correctly formed expressions and interpreted as "searching," "addition," "comparison," etc. Such an approach allows for replacing dynamic programs by lexicographic expressions (static description).

Now, if the input devices of the computer are singled out and some interpretation of their state is adopted, we get a sublanguage—*input language*—of the computer. It is very important to realize that the language, as defined above, depends on the interpretation rules; thus, the same linguistic system (embodied in hardware) may give rise to many different languages.

Let a function φ be given by a set of expressions of a language L. This set is called a *program of the function* φ *in the language* L. Languages designed for defining (in any sense) functions are called *programming languages*. All languages used currently for actual programming belong to a special type of programming languages, viz., *algorithmic languages*, or *process languages*. The distinguishing feature of these is that a function is given by its algorithm, i.e., a set of prescriptions which, if obeyed consecutively, lead from an argument to the corresponding value of the function.

The algorithmic languages do not, however, represent the only possible structure of programming languages. Another example of programming languages is the language of *primitive-recursive functions*, defined as follows:

Alphabet

(1) Symbol $\mathbf{0}$, called zero, is interpreted as number zero.

(2) Symbol $'$, called dash, has no independent meaning; if, however, $\mathbf{A}$ is interpreted as the number A, then $\mathbf{A}'$ is interpreted as the number A + 1. Thus $\mathbf{0}'$ is interpreted as the number 1.

(3) Symbols $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3, \ldots$, called variables, are interpreted as variables assuming positive integer values.

(4) Symbols $\boldsymbol{\varphi}_1$, $\boldsymbol{\varphi}_2$, $\boldsymbol{\varphi}_3, \ldots$, called functional symbols, are interpreted, depending on context, as names of functions.

(5) Symbols (,), and =, called left parenthesis, right parenthesis, and equality sign, have no independent interpretation, but influence the interpretation of expressions in which they occur.

Syntax

(1) $\mathbf{0}$ is a number.

(2) If $\mathbf{A}$ is a number, then $\mathbf{A}'$ is a number.

(3) Every number is a term.

(4) Every variable is a term.

(5) If $\mathbf{\Phi}$ is a functional symbol and $\mathbf{T}_1, \ldots, \mathbf{T}_n$ are terms, then $\mathbf{\Phi}(\mathbf{T}_1, \ldots, \mathbf{T}_n)$ is a term.

(6) If $\mathbf{T}_1$ and $\mathbf{T}_2$ are terms, then $\mathbf{T}_1 = \mathbf{T}_2$ is an equation.

(7) The equation $\mathbf{\Phi}(\mathbf{X}) = \mathbf{X}'$, where $\mathbf{\Phi}$ is a functional symbol and $\mathbf{X}$ is a variable, is a primitive-recursive description. Its interpretation is the description of the function φ that puts into correspondence to a number A the number $A + 1$. $\mathbf{\Phi}$ is interpreted as the name of the function φ.

(8) The equation $\mathbf{\Phi}(\mathbf{X}_1, \ldots, \mathbf{X}_n) = \mathbf{A}$ is the primitive-recursive description interpreted as the description of the function φ which to any n-tuple of its arguments puts into correspondence the number A. The symbol $\mathbf{\Phi}$ is interpreted as the name of the function φ.

(9) The equation $\mathbf{\Phi}(\mathbf{X}_1, \ldots, \mathbf{X}_n) = \mathbf{X}_i$ is a primitive-recursive description that is interpreted as the description of the function φ which to any n-tuple of its arguments puts into correspondence the value of its ith argument $(1 \leqslant i \leqslant n)$. $\mathbf{\Phi}$ is interpreted as the name of the function φ.

(10) Let (i) $\mathbf{P}_1 \mathbf{P}_2 \ldots \mathbf{P}_m$ be a primitive-recursive description,
(ii) $\mathbf{\Phi}_0 \mathbf{\Phi}_1, \ldots, \mathbf{\Phi}_k$ be functional symbols appearing in it,
(iii) the left-hand side of the equation in which these functional symbols appear for the first time be $\mathbf{\Phi}_0(\mathbf{X}_1, \ldots, \mathbf{X}_k)$, $\mathbf{\Phi}_i(\mathbf{X}_1, \ldots, \mathbf{X}_n)$, $1 \leqslant i \leqslant k$,
(iv) $\mathbf{\Phi}$ be a functional symbol not appearing in the primitive-recursive description under consideration,

then $\mathbf{P}_1 \ldots \mathbf{P}_m \mathbf{P}_{m+1}$, where $\mathbf{P}_{m+1}$ is of the form

$$\mathbf{\Phi}(\mathbf{X}_1, \ldots, \mathbf{X}_n) = \mathbf{\Phi}_0(\mathbf{\Phi}_1(\mathbf{X}_1, \ldots, \mathbf{X}_n), \ldots, \mathbf{\Phi}_k(\mathbf{X}_1, \ldots, \mathbf{X}_n))$$

is a primitive-recursive description that is interpreted as the description of all functions given by the initial description *and* of the function φ obtained by substitution of the functions $\varphi_1, \ldots, \varphi_k$ into the function φ_0, where $\varphi_i (0 \leqslant i \leqslant k)$ are the functions whose names are $\mathbf{\Phi}_i$.

The symbol $\mathbf{\Phi}$ is interpreted as the name of this function φ.

(11) Let (i) $\mathbf{P}_1\mathbf{P}_2 \ldots \mathbf{P}_m$ be a primitive-recursive description,
(ii) $\mathbf{\Phi}_1$ and $\mathbf{\Phi}_2$ be functional symbols appearing in (i) and the left-hand sides of the equations in which they appear for the first time be of the form $\mathbf{\Phi}_1(\mathbf{X}_1, \ldots, \mathbf{X}_{n-1})$, $\mathbf{\Phi}_2(\mathbf{X}_1, \ldots, \mathbf{X}_{n+1})$,
(iii) $\mathbf{\Phi}$ be a functional symbol not appearing in (i),

then $\mathbf{P}_1 \mathbf{P}_2 \ldots \mathbf{P}_m \mathbf{P}_{m+1} \mathbf{P}_{m+2}$, where $\mathbf{P}_{m-1}$ is $\mathbf{\Phi}(\mathbf{0}, \mathbf{X}_2, \ldots, \mathbf{X}_n) =$

$\mathbf{\Phi}_1(\mathbf{X}_2 \ldots, \mathbf{X}_n)$ and $\mathbf{P}_{m+2}$ is $\mathbf{\Phi}(\mathbf{X}'_1, \mathbf{X}_2, \ldots, \mathbf{X}_n) = \mathbf{\Phi}_2(\mathbf{X}_1, \mathbf{\Phi}(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n), \mathbf{X}_2, \ldots, \mathbf{X}_n)$, is interpreted as the description of all functions described by (i) *and* of the function φ which is expressed by the functions φ_1 and φ_2 :

$$\varphi(0, X_2, \ldots, X_n) = \varphi_1(X_2, \ldots, X_n),$$

$$\varphi(X_1 + 1, X_2, \ldots, X_n) = \varphi_2(X_1, \varphi(X_1, X_2, \ldots, X_n), X_2, \ldots, X_n).$$

The symbol $\mathbf{\Phi}$ is interpreted as the name of the function φ.

(12) Let $\mathbf{P}_1 \ldots \mathbf{P}_m$ be a primitive-recursive description and $\mathbf{\Phi}_k$ the functional symbol of the left-hand side of equation $\mathbf{P}_m$. Then the primitive-recursive description $\mathbf{P}_1 \ldots \mathbf{P}_m$ is called a program for the function φ_k, whose name is $\mathbf{\Phi}_k$.

Example. The primitive-recursive description $\mathbf{P}_1\, \mathbf{P}_2\, \mathbf{P}_3\, \mathbf{P}_4\, \mathbf{P}_5\, \mathbf{P}_6$, where

$[\mathbf{P}_1]$ $\mathbf{\Phi}_1(\mathbf{X}_1) = \mathbf{X}_1$,
$[\mathbf{P}_2]$ $\mathbf{\Phi}_2(\mathbf{X}_1) = \mathbf{X}'_1$,
$[\mathbf{P}_3]$ $\mathbf{\Phi}_3(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3) = \mathbf{X}_2$,
$[\mathbf{P}_4]$ $\mathbf{\Phi}_4(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3) = \mathbf{\Phi}_2(\mathbf{\Phi}_3(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3))$,
$[\mathbf{P}_5]$ $\mathbf{\Phi}_5(\mathbf{0}, \mathbf{X}_2) = \mathbf{\Phi}_1(\mathbf{X}_2)$,
$[\mathbf{P}_6]$ $\mathbf{\Phi}_5(\mathbf{X}'_1, \mathbf{X}_2) = \mathbf{\Phi}_4(\mathbf{X}_1, \mathbf{\Phi}_5(\mathbf{X}_1, \mathbf{X}_2), \mathbf{X}_2)$

is a program for the "adding function" $\varphi_5(x_1, x_2) = x_1 + x_2$.

Of course the primitive-recursive function method of programming is not a very convenient one, and serves as an example rather than as an object to be achieved. Nevertheless, the possibility of programming not only by means of algorithms but also by means of relations is shown by this example. Thus, the general conclusion may be stated that a generalized programming language should include not only algorithms, but also relations as standard means of describing procedures. Such a language is proposed in [*41*] and is based on the theory of Markov algorithms and on the method of primitive-recursive functions just described.

There are one or two further points that deserve to be mentioned. The approach to programming described here implies what sort of micro-operations should be included into the code list of a computer in order to facilitate autoprogramming. In fact, Shreĭder [*53*] examined the Soviet three-address computers from this point of view, and suggested that, although they are general purpose computers, many changes in their code lists are necessary, or at least desirable, in order to simplify automatic programming for a given class of problems. This should not surprise the reader, since only the statistical analysis of a large number of problems programmed in languages which describe not only macro-operations but also micro-operations, i.e., the computer itself, makes it possible to decide *which* correctly formed expressions pertinent to the programs should be incorporated into hardware.

Finally, it is worth mentioning that regarding the computer as a language may lead to an entirely new concept of reliability.

1.6 Special Purpose Autocodes

All that has been said so far about automatic programming in the USSR is restricted to general autocodes, i.e., programs that accept a source program written in terms of symbolics, and produce an object program written in machine code, regardless of the content of the source program; provided, of course, that the latter was written according to certain rules. This is true even in the case of Glushkov-type autocodes (cf. Section 1.5). However, it is not difficult to see that in practice a somewhat different situation arises quite often, viz., it becomes profitable to possess a specialized autocode which accepts source programs belonging to a more or less restricted class only. Such an autocode operates very quickly, and occupies comparatively little machine memory. As a matter of fact, the attentive reader has undoubtedly noticed that an autocode of Glushkov type consists of a number of specialized autocodes of just this kind. In the first part of this section we give in more detail an interesting example of a specialized autocode due to O. K. Daugavet and Ye. F. Ozerova [*9*]. This autocode is designed to perform operations on matrices, vectors, and scalars and thus is applicable in all problems of linear algebra.

This autocode produces object programs from source programs in which actual values of data need not be given explicitly. Hence, in a sense, the object program is data-independent and may be used many times with different sets of data, provided the structure of the data is the same in each case.

Information about data structure is presented to the computer in the following form. Each variable (matrix, vector, or scalar number) is given a name, denoted by the programmer as a capital letter (or combination of letters) but afterwards coded as a number, since the computer does not accept alphabetic information. A matrix M which is to be stored as a two-dimensional array in machine memory, starting with location a_1, is represented as

$$\begin{array}{cccc} M & a_1 & m & h_1 \\ & & n & h_2, \end{array}$$

where m and n are the dimensions of the matrix, and h_1 and h_2 denote the "step length" in the array in the directions corresponding to the dimensions m and n. A p-dimensional vector V, located from a_2, is represented by

$$V \quad a_2 \quad p \quad h,$$

where h is the step length, i.e., the number of cells occupied by one component of the vector. A number located in a_3 is represented by

$$N \quad a_3 \quad 0 \quad 0.$$

The symbolical inscription

$$N \quad A \quad k_1 k_2 0\, 0$$

denotes an element of matrix A, the address of which is defined (inside the array representing matrix A) by the relation $a_N = a_1 + \bar{k}_1 h_1 + \bar{k}_2 h_2$, where the bars indicate that the dynamically last (i.e. most recent) values of the parameters k_1 and k_2 are taken into consideration.

The parameters may be either constant or variable, in the latter case they may be mutually dependent. Formulas of the type $\bar{k} = \alpha \bar{k}_i + \beta \bar{k}_j + \gamma$, where α, β, and γ are numbers, define actual values of variable parameters.

The program itself is very simple and consists of sequences of "commands" written in the form $n_i ABC$, where n_i is a pseudo-code number denoting the operation to be performed on the variables A, B, and C. If, e.g., n_i assumes the value of the pseudo-code denoting matrix multiplication, the command is equivalent to multiplication of matrices A and B and storing of the result in the location assigned to matrix C. An alternative form of commands is

$$\begin{array}{ll} n_j & A\ F\ R \\ & X \end{array}$$

This form is used when the result of matrix operation n_j is a vector, as happens, e.g., if n_j is the pseudo-code for solving simultaneous linear equations, A being the associated matrix of the system, F the column vector of right-hand expressions, X the result vector, and R a column vector of working cells.

There is one feature of the autocode which deserves special attention. Some of the variables may be marked by a special sign (following the appropriate capital letter) which means that the marked variable will never be used in consequent operations. It is clear that the translator (or rather compiler) interprets this information in such a manner that storage occupied by the marked variable may be used for variables (or intermediary results) if they are generated after (in the dynamic sense) the execution of the command containing the marked variable. Additionally, part of the information may be incomplete, i.e., need not include initial locations. Such variables will be located either in storage previously occupied by "marked variables" or, if these locations are still occupied, in any other locations available.

The Daugavet-Ozerova autocode is, in a sense, an abbreviated form of a partial programming program of Glushkov type. Next we describe an interesting nonnumerical autocode due to Shurygin and Yanenko [*56*]. The autocode is devised to perform some simple, yet very useful, operations on algebraic expressions written in alphanumeric form. The expressions accepted by the autocode are of the type

$$P = \sum \pm \, a_i \, x_1^{\alpha_{1i}} \, x_2^{\alpha_{2i}} \, \dots \, x_n^{\alpha_{ni}}, \tag{1.6.1}$$

where α_{ki} and a_i are real numbers that may be recorded in machine memory and x_k are letters. If x_j is a variable it may be a function of other variables. Expressions like (1.6.1) are called *polynomials*. Observe that this use of the word is more general than the customary one, in which the exponents must be non-negative integers. A large class of expressions used in mathematics can be represented as polynomials in this sense, e.g., the expression

$$R = \frac{ax + \sqrt{cy^2 + 1}}{dx^2 + ey^2} \tag{1.6.2}$$

may be replaced by the polynomial

$$R = axu^{-1} + v^{1/2}\, u^{-1}, \tag{1.6.3}$$

where $u = dx^2 + ey^2$, $v = cy^2 + 1$.

The source program consists of the data description and the program itself. The data description, in turn, consists of two parts. In the first part all pertinent polynomials are coded as pseudo-instructions. (This would be unnecessary if alphabetic symbols were accepted by the computer). The second part gives code numbers for polynomials treated as entities; i.e., each polynomial is given a code number serving henceforth as identifier of this polynomial.

The sequence of operations to be performed on the polynomials is laid down in the program, formally similar to a conventional program written in machine language; the pseudo-instructions are built up from polynomial identifiers and pseudo-codes of operations performable on polynomials by the autocode. The list of performable operations includes:

(1) Replacement of a letter in a polynomial by another polynomial. Let the polynomial P contain nonnegative integer powers of the letter

q, while the polynomial Q does not contain the letter q at all. Then it is possible to substitute Q into P in place of q. For convenience an additional convention is introduced, viz., the code number of the polynomial Q is identical with the code number of the letter q. Then, if one wishes to replace $q_1, q_2, \ldots, q_n$ in P by polynomials $Q_1, Q_2, \ldots, Q_n$, and the code numbers of letters q_{i+1} and q_i differ by one, it is sufficient to specify in the instruction P, Q_1, and Q_n only. Such a complex substitution is possible even in the case when P does not contain all of the q_i $(1 \leqslant i \leqslant n)$, or when not all of the Q_i $(1 \leqslant i \leqslant n)$ have been given in the data description, and thus not all of them "exist." The autocode will choose in this case those pairs (q_i, Q_i) for which substitution is possible.

Addition, subtraction, and multiplication of polynomials is performed by means of the *replace* instruction, described above. For this purpose special standard polynomials, $A = x + y$, $S = x - y$, and $M = xy$, are introduced.

Replacement may be performed either with, or without, reducing similar terms, since the reduction takes up a long time and is not necessary in intermediary polynomials.

(2) Differentiation of polynomials. Let the polynomial R be a function of $x_1, \ldots, x_n$, $t_1, \ldots, t_m$, and let $x_1, \ldots, x_n$ be differentiable functions of $t_1, \ldots, t_m$. With a given table of derivatives

$$\frac{\partial x_i}{\partial t_k} = y_{ik}, \qquad i = 1, 2, \ldots, n, \quad k = 1, 2, \ldots, m, \tag{1.6.4}$$

the autocode finds any of the polynomials

$$S = \frac{\partial R}{\partial t_k} = \sum_{i=1}^{n} \frac{\partial R}{\partial x_i} y_{ik} + \frac{\partial^* R}{\partial t_k}. \tag{1.6.5}$$

Of course, S is recorded in the form (1.6.1). In (1.6.5) $\dfrac{\partial^* R}{\partial t_k}$ denotes the partial derivative with reference to t_k entering R explicitly.

Besides these two fundamental operations, there are some others, of an auxiliary character.

(3) Rename the polynomials $P, P + 1, \ldots, P + n$ by $Q, Q + 1, \ldots, Q + n$.

(4) Duplicate the polynomials $P, P + 1, \ldots, P + n$ as polynomials $Q, Q + 1, \ldots, Q + n$.

(5) Read polynomials $P, P + 1, \ldots, P + n$.

(6) Print polynomials $P, P + 1, \ldots, P + n$.

(7) Erase polynomials $P, P + 1, \ldots, P + n$.

In operations 3 through 7, n may assume the value zero, in which case the operation is performed on the single polynomial P.

This polynomial-handling autocode was applied by the original authors to various problems related to partial differential equations of hydrodynamics and to the extremely tedious work of evaluating determinants consisting of alphanumerical elements.

Yanenko [*69*] gives general rules for reducing systems of quasilinear differential equations:

$$a_i \frac{\partial u}{\partial x} + b_i \frac{\partial u}{\partial y} + c_i \frac{\partial v}{\partial x} + d_i \frac{\partial v}{\partial y} + e_i \frac{\partial w}{\partial x} + f_i \frac{\partial w}{\partial y} = 0, \qquad (1.6.6)$$

$$i = 1, 2, 3, 4,$$

where $a_i, \ldots, f_i$ are functions of u, v, and w, to the form of a single quasi-linear differential equation for the function $w = w\,(u, v)$. These rules were programmed for the STRELA computer and many checks were conducted. In [*56*] an example is given of a set of three differential equations of the form (1.6.6) reduced to one differential equation in 12 minutes by the computer, using this automatic coding system.

Shurygin and Yanenko [*56*] give a detailed description of the technical features of this specialized autocode, which are of certain interest to students of similar problems, but will not be discussed in the present article.

1.7 Use of Topology in Programming

In this section we discuss a paper which provides an example of the application of very advanced mathematical apparatus to practical problems of automatic programming. Lavrov [*40*] considers the question of economy of machine memory in closed operator schemes. Following his argument we shall see how this problem is reduced to a distribution problem, closely related to the famous problem of graph colors.

Let us consider two sets of entities: quantities x_j and operators S_i. There are two possible relations between an operator and a quantity, if they are related at all. The operator may either *use* the quantity, or *generate* it. In the first case the quantity is called *argument* of the operator, in the second case the *result* of the operator. Needless to say, a quantity which is an argument of one operator may be the result of another.

Between two different operators the relation called *transition* may occur, in which case one operator is called *predecessor*, the other *successor*.

An operator scheme is defined by two finite sets $\mathbf{X} = \{x_1, \ldots, x_n\}$

and $\mathbf{\Sigma} = \{S_1, \ldots, S_m\}$ and three matrices $\mathbf{A} = [a_{ij}]$, $\mathbf{B} = [b_{ij}]$, and $\mathbf{C} = [c_{jk}]$, where $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, m$; $k = 1, 2, \ldots, m$. The matrices are Boolean with the following rules for non zero elements:

$a_{ij} = 1$ if, and only if, S_j uses x_i.
$b_{ij} = 1$ if, and only if, S_j generates x_i.
$c_{jk} = 1$ if, and only if, there is a transition from S_j to S_k.

In addition to this we shall consider a few other concepts. A *link* in the scheme is any pair of integers (j, k) such that (i) $1 \leqslant j \leqslant m$, $1 \leqslant k \leqslant m$ and (ii) $c_{jk} = 1$; a *path* in the scheme is any ordered sequence of positive integers $(j_0, j_1, j_2, j_3, \ldots, j_s)$ such that any pair (j_{k-1}, j_k) with $1 \leqslant k \leqslant s$ is a link; thus a link is a special case of a path. An *itinerary* of the quantity x_i is a path $(j_0, \ldots, j_s)$ in which the operator S_{j_s} uses x_i and none of the $j_1, \ldots, j_{s-1}$ operators generates it. The itinerary is *closed* if S_{j_0} generates x_i, otherwise the itinerary is called *open*. The operator numbered j_0 is called initial operator of the itinerary, and S_{j_s} its terminal.

We shall say that the operator scheme is closed if each open itinerary of the quantity x_i belongs to at least one closed itinerary of the same quantity.

The ordered pair (x_i, S_j) denotes the value of the quantity x_i before execution of the operator S_j; the pair (S_j, x_i) denotes the value of the quantity x_i after execution of the operator S_j. The pairs (S_{j_0}, x_i), (x_i, S_{j_1}), $\ldots$, (x_i, S_{j_s-1}), (S_{j_s}, x_i) are said to belong to the itinerary $(j_0, j_1, \ldots, j_s)$. From the set of all $2mn$ possible pairs associated with the operator scheme we single out a subset, called *encumbrance* set, which consists of (i) pairs (x_i, S_j) such that S_j is either internal or terminal in a closed itinerary of x_i, (ii) pairs (S_j, x_i) such that S_j is either internal or initial in a closed itinerary of x_i.

At this point the reader deserves a short break in the continuous flow of definitions. This shall be achieved by a few almost trivial remarks. The concepts introduced above bear close resemblance to the usual notions of programming. Quantity is a generalization of register content; operator corresponds to instruction; transition is similar to transfer of control; and operator scheme is very close to program. A closed itinerary is a sequence of instructions such that the first one generates the content of a register x_i and all but the last of the other instructions of the sequence use the value recorded in the register; in other words, the register x_i is occupied by the same value during the execution of the sequence (cf. region of existence, described in section 1.3).

If the pair (x_i, S_j) belongs to the encumbrance set, then the register

x_i is occupied *before* the generalized instructions S_j is obeyed. If the pair (S_j, x_i) belongs to the encumbrance set, then the register x_i is occupied *after* S_j is obeyed.

Now let us consider generalized readdressing. For this purpose we take the scheme $\mathbf{\Sigma}$, $\mathbf{X}$, and the set $\mathbf{Y} = \{y_1, \ldots, y_n\}$. Let M denote a subset singled out of the set of all $2mn$ pairs of the scheme. We denote by F the function which satisfies the conditions: (i) F is defined for all pairs belonging to M and (ii) the values of F belong to $\mathbf{Y}$. The function F is called *readdressing function.* We shall say that S_j uses $y_{i'}$ if and only if, S_i uses x_j and $F(x_i, S_j) = y_{i'}$. Similarly, if and only if S_j generates x_i and $F(S_j, x_i) = y_{i'}$ we say that S_j generates $y_{i'}$.

Of particular interest are functions F which satisfy the additional conditions:

$$\begin{array}{ll}\text{For given } j, & \{(x_{i_1}, S_j), \ldots, (x_{i_k}, S_j)\} \in M, \\ \text{and for } 1 \leqslant \alpha < \beta \leqslant k & F(x_{i_\alpha}, S_j) \neq F(x_{i_\beta}, S_j).\end{array}$$

If F satisfies this condition we say that the new notation system $\mathbf{Y}$ is consistent on input of S_j. Similar conditions must be satisfied if $\mathbf{Y}$ is to be consistent on output of S_j. We call $\mathbf{Y}$ *consistent* if it is consistent on both input and output of all $S_i \in \mathbf{\Sigma}$.

If M includes all pairs that belong to the encumbrance set, the new notation is called *complete*. If each pair of M belongs to the encumbrance set we call the new notation *minimal*. In other words, for a complete and minimal set, M is identical with the encumbrance set. Finally, if for a complete, minimal, and consistent notation system, and for any itinerary $(j_0, j_1, \ldots, j_s)$ of the quantity x_i

$$\begin{aligned} F(S_{j_0}, x_i) &= F(x_i, S_{j_1}) = F(S_{j_1}, x_i) = \cdots = F(x_i, S_{j_{s-1}}) \\ &= F(S_{j_{s-1}}, x_i) = F(x_i, S_{j_s}), \end{aligned} \tag{1.7.1}$$

the new notation is said to be *equivalent* to the old one.

With the help of the definitions introduced so far we may formulate the following problem: For a given closed operator scheme $\mathbf{\Sigma} = \{S_1, S_2, \ldots, S_m\}$ and $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$, find a notation system $\{y_1, \ldots, y_{n_0}\} = \mathbf{Y}$, equivalent to $\mathbf{X}$, such that for any other notation system $\{z_1, z_2, \ldots, z_{n_1}\} = \mathbf{Z}$ which is equivalent to $\mathbf{X}$, the inequality

$$n_1 \geqslant n_0 \tag{1.7.2}$$

is satisfied.

It is easy to see that this problem is of utmost importance in the theory of programming. The most difficult part in the solution of the stated problem is the equivalence criterion. The given definition of equivalence demands that some conditions should be satisfied for *all* closed itineraries. Since the number of itineraries is infinite and they are not easily ordered, the equivalence criterion should be reformulated in such a manner that it would not involve infinite sets.

We define the *bearer* of the quantity x_i as subset of the encumbrance set, such that each element of the bearer contains x_i and all elements of M that contain x_i are included in this subset. The bearer of x_i will be denoted by M_i.

Let us consider two pairs m and m'. We shall say that $m, m' \in M$ are *connected* if there exists either a closed itinerary of x_i to which both m and m' belong, or a third pair $m'' \in M$, that is connected with both m and m'. The bearer M_i is connected if all of its elements are connected.

M_i falls into subsets (with empty common parts) such that (i) any two pairs of the same subset are connected and (ii) no two pairs belonging to different subsets are connected. These subsets are said to be *regions of activity* of x_i.

If M_i is connected it consists of exactly one region of activity; in the general case the bearer M_i consists of a finite number of regions of activity.

Lavrov has proved the following:

Theorem. The new consistent system $\{y_1, y_2, \ldots, y_{n_1}\}$ is equivalent to the old one, $\{x_1, x_2, \ldots, x_n\}$ if and only if the readdressing function F is defined on the encumbrance set, and is constant inside of each of the regions of activity of arbitrarily chosen x_i.

Proof. Since the readdressing function F is defined on the encumbrance set M, the new system $\mathbf{Y}$ is complete and minimal. It remains to be shown that if, in each region of activity, $F =$ constant then along any closed itinerary the equality (1.7.1) holds true, and vice versa.

Proof of necessity. Let the pairs $m \in M$ and $m' \in M$ belong to the same region of activity, then, by definition, there is a sequence $m_0 = m$, $m_1, \ldots, m_k = m'$ such that, when $j = 1, 2, \ldots, k$, the pairs m_{j-1} and m_j belong to the same closed itinerary; but since the systems are assumed to be equivalent, $F(m_{j-1}) = F(m_j)$, $j = 1, 2, \ldots, k$, which shows that for arbitrarily chosen pairs m and m' belonging to the same region of activity the value of F is the same. This proves the necessity part of the theorem.

Proof of sufficiency. Let $(j_0, j_1, \ldots, j_s)$ be an arbitrarily chosen itinerary of x_i. All pairs belonging to it belong to the same region of activity x_i, thus Eq. (1.7.1) is fulfilled. This completes the proof of the theorem.

In this way Lavrov reduced the problem of equivalence to the problem of determining regions of activity. But this problem may be solved easily for any operator scheme by means of an algorithmized process. An ALGOL procedure for this algorithm is enclosed as an appendix to Lavrov's paper.[11]

1.8 "Philosophy" of Automatic Programming

Soviet academician Sobolev, leader of the Siberian Division of the Mathematical Institute of the Academy of Sciences, has stated [*57*] that the time is coming when automatically produced programs will in every aspect outdo manual programming. This view, contestable as it is, faithfully represents the "philosophy" underlying Soviet research in this branch of science. We leave the discussion of Sobolev's opinion to Sections 2.1 and 2.3.1 and restrict ourselves in the present section to an examination of the motivation and perspectives of automatic programming in the USSR.

An obvious stimulus for the automation of programming for electronic computers is the ever-growing ratio of the number of computers produced to the number of qualified staff trained. This problem becomes very acute even in the USSR (cf., e.g., [*36*]), in spite of the fact that a considerable yearly output of numerical analysts acquainted with computers impaired the demand for programmers during the early years of electronic computing. Now, however, the situation is changing. Computers are becoming more and more popular and are entering into so many branches of life that trained mathematicians, no matter how large their number, cannot cope with the programming of computers for various, and quite often small, problems. Besides, it would be a waste of highly skilled labour to use it for programming on a large scale.

Automatic programming makes the programming process easier and quicker for numerical analysts, and facilitates programming of computers by specialists not necessarily having any numerical background —by biologists, chemists, and others.

But this is not the only reason for the intensive search for better and better autocodes. A second, and seemingly more important, one is given in Sobolev's statement. This second reason is the persistent belief that automatic programming may be not only faster but also better than manual programming.

Fedoseev [*18*] regards the manually produced programs as possessing, inherently and unavoidably, mistakes and errors that need not affect

[11]The procedure is written in the Russian version of ALGOL, but an enclosed dictionary makes it possible to read this procedure, except comments, without any knowledge of the Russian language.

the results of calculation but make the computation process unnecessarily long and complicated. The reason for this is the lack of lucidity of programs thus produced, and the extremely difficult and time-consuming procedures which have to be applied in order to check the program against the original algorithm; for the generally accepted method of checking programs is either to have two programs doing the same job and compare the results of calculations, or to have one programmer checking thoroughly somebody else's programs. This should make it clear why almost all Soviet theoreticians consider built-in checking procedures as a *conditio sine qua non* of a good automatic programming system (cf. Section 1.2).

In Section 1.5 we considered the automatic programming system proposed by Shreider. He claims that the way leading to better and fuller automation of programming goes through the revision of traditional lists of operations microprogrammed for digital computers. In the same section we gave some ideas about Glushkov's proposals. Generally, however, it is assumed that the best path to the automation of programming is that laid down by Lyapunov and his collaborators—the method of programs which program. It is perhaps interesting to analyze the criteria of the "goodness" of PP's, which are given in [*18*].

The qualification of a PP manifests itself in its ability to accept as many common mathematical symbols and conventions as possible. A qualified PP should "understand," if needed, omitted parenthesis, priority of operations, a minus sign preceding an open parenthesis, etc. On the other hand, the more qualified the PP the more should it take advantage of specific features of the computer, use built-in cycles, readdressing facilities, and so on. An important job for a qualified PP is to economize the object codes. This may be done either by scanning through a rough version of the object program, or by picking out similar expressions in the source program. The second way gives better results, though it takes more time. Automatic storage allocation is another fundamental feature of a qualified PP.

Speed of PP is not considered an equally important factor, since in most cases programming by PP takes only a tiny fraction of computing time. Certainly, acceleration of the PP at the cost of reduced qualification is not worth considering.

A very important factor influencing the goodness of the PP is *data structure* and *presentation*. Compactness of acceptable data sets is one of the desirable qualities; but more fundamental demand is that the data should be local, i.e., should not depend on the structure of machine memory, levels of storage, and so forth. A good PP allows for rapid changing and replacing of one data set by another. This is essential both in production runs of the computational program and in its trial runs.

Further, a good PP should not impose any restrictions on the *length* of the resulting *object program*, except, of course, "natural" limitations, such as machine memory capacity. This implies, among other things, that the PP should provide for transferring previously produced sections of the object program to the backing store.

There are two secondary criteria of goodness, viz., *length* of the programming program and *simplicity* of methods used and notation adopted.

Several authors emphasize that a good PP should of necessity include self-checking and automatic checking facilities both for translation and actual calculations. (We may mention here that devices like internal parity checking are not common in Soviet computers).

Finally, Fedoseev makes interesting remarks on the effects which employment of a PP has on the staff of a computing laboratory. He finds that the use of programming programs leads to natural division of the programmers employed by the laboratory into two groups—programmers concerned with problems to be solved by the computer and programmers concerned with the PP. The first group gradually loses interest in technical details of programming and may subsequently become indifferent to questions like the possibility of saving computing time by, say, foregoing the use of some standard subroutines available through PP. The second group may, reversely, lose any interest in the mathematical meaning of the programmed problems and devote themselves completely to the upkeep of the PP. It is perhaps worth mentioning that in Fedoseev's opinion a PP becomes obsolete much sooner than the computer itself, and thus the necessity arises of continuous improvement in the existing programming program.

Bibliographical notes. To the best of my knowledge, there is only one paper [*18*] in the relevant Soviet literature devoted to a general discussion of methods of automatic programming. Many interesting remarks are, however, to be found in some papers referred to in preceding sections, especially in [*41*], [*19*], [*53, 54*], [*27*], [*38*]. Also, I would like to bring to the reader's attention a textbook on programming [*37a*, in Russian], which covers much of the pertinent work done in the USSR and contains a rather ample bibliography. Additionally, the following papers may be recommended: [*15*], [*28*].

2. Poland

2.1 Historical Remarks and Underlying Philosophy

In Poland there are two organizations concerned with automatic programming: the Institute of Mathematical Machines and the Computation Center, both being divisions of the Polish Academy of Sciences. These two organizations differ in respect to their roles and to the scope of their work, as well as in their histories and their general approach to the problem of automatic programming. IMM, which is an offshoot of the Mathematical Institute of PAS, is primarily concerned with designing and producing electronic digital computers. As a matter of fact, three computers were designed there: SKRZAT, XYZ, and ZAM-2. Several copies of the latter have already been produced by the IMM and are being operated in various computing laboratories; the two former types of computers were rather experimental. The CC was set up in September 1961; a large part of its staff previously formed the Applied Mathematics Department of the Nuclear Research Institute. It is primarily concerned with practical applications of computers in various branches of science and in economics, and operates a number of different computers, including the first URAL-2 installed in Poland. The formal division of work does not encompass all the research done by these organizations, since, e.g., one of the first Polish computers, EMAL-2, was constructed under Dr. R. Marczynski's guidance at CC, and IMM conducts extensive research in applications of computers to banking and administration problems.

As far as the questions of automatic coding are concerned, both institutions are doing equally pioneering jobs: IMM has constructed three systems of external programming for their family of computers, and CC is working on alphanumeric codes for the Soviet-made URAL-2 computer.

On comparison of the work already accomplished by these two institutions and of the attitudes taken by the authors of published papers representing the points of view of the automatic coding groups in IMM and CC, a very curious conclusion can be reached. The IMM people consider the possibilities of automatic programming to be limited by the structure of existing computers and, as stated in [*44*], are of the opinion that ". . . a practical realization of a fully universal language, independent of all individual machine features, will always lead to compromises at the expense of the full economical use of the machine." In other words, though realizing the neccessity of the external language to take full advantage of all machine potentialities, they still find that, in practice, automatic coding does not exploit these potentialities to the maximum, thus leaving it to the manual coding to produce the most economical programs. At the same time, the automatic programming group in IMM does not seem to conduct any research on the problem of how to change the traditionally accepted lists of operations microprogrammed in the computer in such a way that more economical use of automatic programming becomes possible.

The automatic programming group at the CC, on the contrary, considers the problem of economical automatic programming to be closely related to the list of operations of the computer, and suggests which of the usually microprogrammed operations are useless from this point of view, and which ones are lacking in existing computers. This opinion is to be found, e.g., in [*22, 23*] and developed in [*21*]. Since, however, the CC does not, at present, build its own computers, and existing ones are not quite fit for optimal translation of source programs into object programs, the Center has taken a somewhat unusual approach, viz., a quasi-manual coding technique has been adopted (cf. Sections 2.3.1 and 2.3.2).[12]

At an International Conference held in Warsaw in 1961, the two groups took different positions and expressed different attitudes towards ALGOL and similar formal external languages. The IMM people, starting from the principle of an ever-persistent tendency to improve any universal language, rejected the proposed freezing of ALGOL, and did not recognize any need for a uniform external language that may be "understood" by all computers above certain limits of speed and memory.

On the other hand CC, though disagreeing with many features of ALGOL, accepted it as a basis for a future external programming system for *all* computers installed in the Center. Furthermore, the version of ALGOL worked out at CC as a hardware representation for the URAL-2 is intended to become the common programming language for all computers of this type installed in the USSR and other countries of Eastern Europe.

In seeking the reasons for these differences, one easily finds that the IMM, operating only self-made computers and incorporating in any new coding system all systems previously used, has already achieved, in a sense, the desirable state of affairs where any program may be run on each of the computers without any additional recoding. On the other hand the Computation Center, operating many different computers, is in the position familiar to all the people trying to run their programs on different computers without a uniform programming language "understood" by all the machines.

In order to complete the general information about automatic programming groups in Poland, we may add that the Institute of Mathe-

[12]It may be worth mentioning that some of the general ideas concerning the design of computers suitable for automatic coding are incorporated in the logical design of the ODRA 1003 computer, produced by the ELWRO factory in Wroclaw, Poland.

matical Machines is at present working on a COBOL-like language for the ZAM computers.

2.2 Work of the SAKO Group

In order to distinguish between the two pertinent centers of research on automatic coding in Poland, we shall use the abbreviations SAKO group and KLIPA group, as convenient nick names for the autocoding groups working at IMM and CC, respectively (cf. Section 2.1).

2.2.1 The SAS Programming System

The SAS programming system was devised, by the SAKO group led by Professor L. Lukaszewicz in 1959—1960, as an external programming language for the XYZ computer [*61*], and afterwards, somewhat enlarged and modernized, adapted for use with the ZAM-2 computer [*17*]. We shall consider this final version only, for the minor changes introduced in it are not important, and any correct "old SAS" program is certainly correct in the new version.

The SAS is a typical semi-autocode. It frees programmers from manual allocation of storage space, allowing for symbolic address parts of instructions; the operational part of an instruction is preserved in machine form, i.e., consists of two letters being the abbreviation of the Polish names of instructions. In addition, SAS allows for automatic inclusion of subroutines, recorded in auxiliary store, into programs, division of programs into chapters, and simplifies the presentation of numerical values to the computer, a feature which is particularly rewarding since the ZAM computer is essentially a fixed-point one, without floating-point facilities other than simplified input of non-standardized numbers.

SAS greatly reduced the number of blunders commonly made by programmers when forced to insert real addresses into the address parts of machine instructions, and consequently reduced the time needed for checking of programs. Furthermore, should any change in a program written in SAS occur, it would not cause readdressing of all instructions (as is necessary when absolute addressing is preserved). In such cases it suffices to change a small number of symbolic addresses only.

SAS uses all but one of the symbols available on standard Creed teleprinter equipment, the omitted one being the £ sign.[13] A SAS program consists of *blocks*, i.e., pieces of program recorded in consecutive

[13]Among the characters available in "figure shift" there are two symbols $\times$ and n to be distinguished from the letters X and N.

registers. Blocks may by divided into *numerical*, containing numbers, and *instructive*, containing commands. The *address* of the beginning of a block is the first register occupied by the block. Symbols used to identify blocks and particular registers may consist of at most four characters, and fall into two categories: *variable symbols* used to identify blocks of numbers, particular numbers and working cells must begin with a letter; *symbolic numbers* used as identifiers of instructive blocks (or labels) begin with digits or the symbol *n*.

A SAS chapter may be divided into *paragraphs*, identified by symbolic names consisting of not more than 3 letters preceded by the * sign. Paragraphs are similar to ALGOL blocks in that symbols used within a paragraph may have quite different meanings in another one. Symbols preceded by the "×" sign are understood as *global symbolic numbers*, which do not lose their meaning on passing from one paragraph to another.

An instruction takes the form:

⟨operational part⟩⟨. or empty⟩ ⟨address part⟩.

⟨operational part⟩ is one of the machine-acceptable symbols for operations. If the address part is preceded by a dot it means that the address is that of a double-length register, there being two word-lengths in the ZAM-2 computer: 18 and 36 bits. ⟨address part⟩ may be either *absolute*, i.e. an unsigned decimal integer, corresponding to the register number, or *relative*, preceded by a "+" or "−" sign, and understood as the address of a word following (+) or preceding (−) the instruction in which they occur, or *symbolic*, in which case it consists of a block identifier supplemented by an integer enclosed in parentheses to denote relative position of the word in the block (if needed). Symbolic addresses that are paragraph identifiers, global symbolic numbers or symbolic numbers, must be followed by right-hand parenthesis.

Numerical values may be written in SAS programs in one of the following forms:

(i) Short integers, i.e. integers occupying half-word registers, are written in the form ⟨sign⟩ ⟨integer⟩, and must not exceed 2^{17}-1.

(ii) Long integers, occupying 36 bit words (one bit being taken up by the sign of the integer) are written in the form: ⟨sign⟩ ⟨integer⟩D, e.g., + 75D, − 276543489D, and must not exceed 2^{35}-1.

(iii) Short numbers, written in the form ⟨sign⟩ ⟨integer⟩ · ⟨decimal fraction⟩ K, occupy half-word registers. If the number is to be recorded in machine memory with the decimal point in another position than that which follows from decimal notation (binary scaling), a corresponding scaling factor is written after the decimal fraction.

(iv) Long numbers are written in the form ⟨sign⟩ ⟨integer⟩·⟨decimal fraction⟩; this may be followed by comma and scaling factor, if binary rescaling is needed. The long number occupies a 36 bit register.[14]

In (iii) and (iv), in the case of too many digits, i.e., when there are more significant digits and "scaling zeros" than 17 or 35, the least significant bits are "rounded off."

Binary scaling, mentioned above, is an essential feature both of the SAS and the ZAM-2 computers, since the latter has no floating point arithmetic except interpretative routines.

Octal numbers preceded by = sign are interpreted as logical scales (cf. Section 1.1) and thus may be either 6 or 12 digits long, depending on the word-length chosen.

Absolute numbers are decimal numbers preceding instructions and separated from them by a solidus. These are interpreted as internal memory locations in which an "absolutely numbered" instruction is to be recorded. Instructions without absolute numbers are recorded in consecutive locations, following the last numbered instruction.

Symbolic numbers are symbolic names given to blocks and separated from the first instruction of a block by a right-hand parenthesis. If the parenthesis is followed by another one, as, e.g., in 291)) OD.CH2, the instruction is located in a half-register with even address; the odd half-register is filled with a "do nothing" instruction.

It is possible to give several symbolic numbers to the same instruction, as in the following example: 4AR)3KOS)UA 512. The combination 342/3KOL)OD 1022 is understood as the first instruction of a block which has the symbolic label 3KOL and which is to be recorded in registers with absolute addresses starting from 342.

The labeling system adopted in SAS is beyond any doubt very convenient, especially when full advantage is taken of paragraphs and chapters, but it imposes extremely heavy requirements on memory space occupied by directories and dictionaries during the translation and, alas, execution of programs.

The last thing to be mentioned about the system is the way in which the standard subroutines are called in. This is done by a *directive* FUN N, where N stands for the number of the desired subroutine. This directive causes the subroutine to be recorded in the body of the chapter in which the directive has been used, starting from the location that

[14]The ZAM-2 computer is an essentially fixed point computer; thus, in machine memory all numbers are recorded as integers. The position of the decimal point and the value of the scaling factor carry information that is used by input (read) routine, and afterwards lost. This information determines how many zeros are to precede first significant digit of the number recorded in the machine memory.

would be occupied by an instruction having the same lexicographic position as the directive FUN *N*. There are some other directives in SAS, i.e., instructions leading to red-tape operations during the translation process. Their meaning is usually self-evident; e.g., START *n*/*N*, CHAPTER *N*, and so on.

We have not given a more detailed description of SAS for two reasons: (i) Section 2.3, describing the KLIPA system, covers the same topic in many respects; (ii) a better example of work of the SAKO group is the SAKO system itself.

2.2.2 The Automatic Coding System SAKO

SAKO was constructed at IMM almost simultaneously with SAS and is used for the programming of the XYZ and ZAM-2 computers, providing for full automation of the coding. It is outside the scope of the present article to give a detailed description of the system, for the manual produced by IMM [*45*] contains some hundred pages, and the reader is perhaps more interested in general features than in details that may be found in technical papers. Thus, hoping that in case of misunderstandings the original literature (see bibliographical note at the end of the current section) will be consulted, we shall begin with short examples showing the general structure of the language SAKO.

(1) Arithmetic formulas:

```
X3 = ALFA + LOG(SIN(X))
 I = I + 2
 S = S + A(I,J) × B(J,K)
 D = 45.678
```

Numbers in parentheses denote running indices of array elements. If, however, the left-hand parenthesis is preceded by a functional identifier (like SIN or LOG), the number in parentheses is interpreted as an argument of the function.

(2) Boolean formulas:

```
  A ≡ B + C
SDF ≡ 0000.4567.3241 × A
```

(3) Control transfers:

```
JUMP  TO  5F
JUMP ACCORDING TO ALFA TO:  3AB,4,2,5C
```

this instruction means:

$$\text{if ALFA} = \begin{cases} 1 \\ 2 \\ 3 \\ 4 \end{cases} \text{jump to} \begin{cases} 3\text{AB} \\ 4 \\ 2 \\ 5\text{C} \end{cases}$$

```
REPEAT FROM 3:   ALFA = 0 (1) 29
GO TO CHAPTER:   GAUSS-SEIDEL
RETURN
IF OVERFLOW:   4 ELSE 3
```

These examples[15] show the flexibility and potentialities of SAKO. Now we shall consider in some detail the most interesting features of programming in this language.

Handling multidimensional arrays. Multidimensional arrays handled by SAKO are called blocks. Each block has to be declared at the beginning of the chapter in which it appears. Block declaration is performed by the statement: BLOCK $(n, m, \ldots, k, r, t)$: ⟨list⟩. The list consists of the names of all the blocks of the chapter having the same structure defined by the contents of the parentheses. The structure of the block is defined as follows: let there be s numbers inside the parentheses following the word BLOCK, then the block declared is an s-dimensional array. Thus, referring to an element of any of the blocks entering the list, we must specify s subscripts (running indices). The numerical values of the s numbers appearing inside of the parentheses in the block declaration represent upper bounds of the subscripts. Hence, the declaration BLOCK (3, 3): MATRIX A, MATRIX B says that in the chapter where the declaration has occurred there are to be reserved registers for two square matrices of 9 elements each. Elements of the matrices are called for in arithmetic statements by expressions like the following: MATRIX A (j, 2), MATRIX B (1, 2). Letters acting as values of subscripts must be declared as integers. The declaration BLOCK reserves space for arrays that must be filled by either reading in or generating in the course of executing the program. If values of elements of the array are to be given in the program itself, another declaration is used, viz., TABLE $(n, \ldots, k)$: ⟨name⟩. This is written

[15]Polish words appearing in SAKO statements are replaced throughout the present section by corresponding English ones.

immediately before the actual values of the elements are explicitly given, e.g.:

```
            TABLE (2, 3): A
3.7195   45.16          .1867
-.876     9.8867   3456.9
```

If the structure of the array is to be changed in the course of calculations, a declaration STRUCTURE $(i, j, \ldots, k, \alpha_1, \alpha_2, \ldots, \alpha_n)$: ⟨list⟩ must be used. In this declaration, $i, j, \ldots, k$ are identifiers of the variables (declared previously as integers) and $\alpha_1, \alpha_2, \ldots, \alpha_n$ are integers; the list has the same meaning as in a BLOCK declaration. The new structure is defined by the items in parentheses in a manner exactly similar to the old structure in the corresponding BLOCK declaration. The total number of elements in the restructured array must not exceed the total number of elements reserved for an array.

A special declaration TABLE OCTAL $(n, m, \ldots, k)$: ⟨name⟩ serves to introduce Boolean arrays.

Boolean expressions. A distinguishing feature of the SAKO system is its ability to handle Boolean words and perform Boolean operations on Boolean words. A Boolean word is a sequence of 18 or 36 binary digits: zeros and ones. In the first case we speak of short Boolean words, in the second of long ones. The words are represented in the octal system, each octal digit representing a group of three consecutive binary digits. For clarity, dots separating groups of octal digits may be used, e.g., a short Boolean word: 707.512, and a long Boolean word: 76.754.324.765.

Boolean words are identified either by individual identifiers, or by subscripted identifiers, when the corresponding word is an element of an octal table. The following operations on Boolean words are available in SAKO:

(1) Renaming of a Boolean word: $A \equiv B$, where A is any identifier, and B is either an identifier or a correct Boolean expression.

(2) Boolean addition: $A \equiv B + C$, where B and C are Boolean words; this operation is the bitwise addition.

(3) Boolean product: $A = B \times C$, where B and C are Boolean words: this operation is equivalent to bitwise multiplication.

(4) Boolean negation $A \equiv -B$, where B is a Boolean word, is the operation defined as bitwise negation of the word B.

(5) Cyclic shift $A \equiv B*(N)$, where B is a Boolean word and N is an integer or integer identifier, is the word A resulting from N cyclic shifts of one position to the right, if $N > 0$, or to the left, if $N < 0$, of the word B.

A correct Boolean expression is formed by application of any number of operations (2) through (5).

Of two operations in a Boolean expression, that one which is enclosed in a greater number of parentheses is executed first.

In case of an equal number of parentheses, the operations are executed with the following priority: (1) shifts, (2) products, (3) sums and negations.

Operations enclosed in the same number of parentheses and of the same hierarchy are executed in sequence from left to right.

To illustrate these rules we take the following example [*44*]. We require the formation of the Boolean product of positions 0 through 9 of the word B and positions 10 through 19 of the word C, add to this the negation of positions 20 through 29 of the word D; the result is to be stored in positions 26–35 of the word A, the remaining positions of the word A being filled with zeros. This may be achieved by executing formula:

$$\mathrm{A} \equiv (\mathrm{B} * (20) \times \mathrm{C} * (10) + (-\mathrm{D})) * (6) \times 000.000.001.777.$$

Additionally, it should be noted that any machine word, hence any number, may be treated as a Boolean word and *vice versa*. This allows for many simplifications customary to programmers working in machine code and inaccessible in a great majority of autocodes.

In SAKO, Boolean words are not used for control transfer operations, as they are in ALGOL.

Input—output operations in the system are conceived to facilitate the programmer's work to the utmost. There are special instructions for reading single numbers, blocks of numbers, captions, octally represented Boolean words and arrays, and so on. The same applies for printing routines, which allow for preparing tables, octal output, and captioning of numerical material.

A program written in SAKO is divided into chapters which facilitate the execution of long programs, exceeding the capacity of the operational memory.

Subroutines and library routines are treated in SAKO in the usual way; many concepts of ALGOL procedures are incorporated, including the formal and actual parameters correspondence rules. There is, however, one important programming device used in the system that bears no resemblance whatsoever to ALGOL, viz., the SUBSTITUTE instruction.

Subroutines are called in by instructions of the form:

$$\langle\text{actual identifiers of results}\rangle = \langle\text{subroutine identifier}\rangle$$
$$\langle\text{identifiers or numbers serving as arguments}\rangle.$$

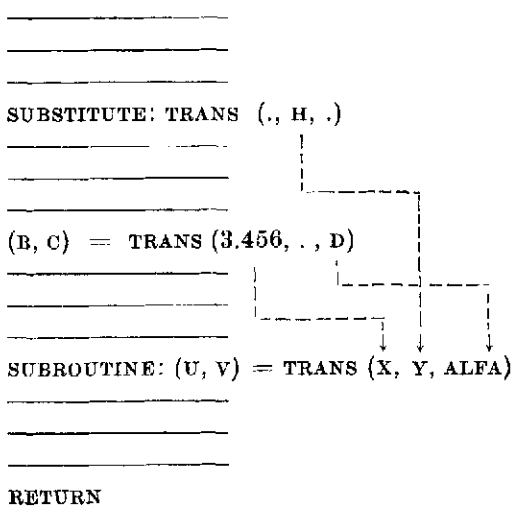
An instruction of the form:

SUBSTITUTE: ⟨subroutine identifier⟩ ⟨partial list of numerical values or identifiers of actual parameters serving as arguments⟩ causes *some* of the formal parameters to be replaced by the values of some of the actual parameters. Those positions on the list which are occupied by the formal parameters that are not to be replaced by actual parameters on execution of this instruction are left empty; technically this is achieved by dots being placed instead of identifiers. Now, when the subroutine call is to be made, the relevant list should have empty (dotted) positions in place of the parameters already inserted by the SUBSTITUTE instructions. The practical value of this trick becomes clear when one takes into account the fact that a subroutine may need to be called by another subroutine, while actual parameters are generated by the main program, and formal parameters (to be replaced on calling by actual ones) are essentially local to the subroutine body.

An example will perhaps add clarity to the above. Consider a subroutine declared as follows:

```
SUBROUTINE: (U, V) = TRANS (X, Y, ALFA)
C = COS (ALFA)
S = SIN (ALFA)
U = X × C + Y × S
V = −X × S + Y × C
RETURN
```

and a program in which irrelevant instructions are replaced by bars:



```
________________
________________
________________
SUBSTITUTE: TRANS (., H, .)
________________
________________
________________
(B, C) = TRANS (3.456, . , D)
________________
________________
________________
SUBROUTINE: (U, V) = TRANS (X, Y, ALFA)
________________
________________
________________
RETURN
```

If the (B, C) = TRANS (3.456, . , D) is used inside of a subroutine, the value of H, computed outside of this subroutine, is inaccessible, thus the only way it as an actual parameter is to use the SUBSTITUTE instruction.

Before closing the section devoted to the work of the SAKO group we will say a little about the translation of programs written in the language of SAKO. The following information is given by Swianiewicz and Sawicki [*62*]. The translation of both SAKO and SAS programs takes place essentially in two stages. The first stage consists in translating the program into a special simplified version of SAS, the so-called SAS-W language. The second stage is the translation of the intermediary code thus produced into a machine code. During translation the SAKO program is read into the machine "one sentence after another." After the sentence has been read in, it is identified and the pertinent translator subroutine is activated in order to translate the sentence into SAS-W (leaving the address part in symbolic form). If the SAKO sentence is labeled, the label is changed into a symbolic number (cf. Section 2.2.1) and recorded in a dictionary. Numbers appearing in arithmetic formulas are translated into binaries. At this stage of translation standard functions, subroutines and library routines are selected (or read in) and a list of them is attached to the SAS-W program chapter. Thereafter, a complete SAS-W chapter is sent to the drum memory. Then, the entire chapter is considered for a second time, symbolic addresses are erased and replaced by real ones, subroutines and functions subjoined, and the resulting program chapter transferred to the drum.

In the translation of arithmetic formulas great care has been taken to optimize the resulting code, and it is stated [*2*] that though the optimization method used has some shortcomings resulting from the need to simplify the translator, a very significant shortening of the resulting program has been achieved.

This high degree of optimization achieved by the SAKO translator is, unfortunately, paid for by low speed of translation. In another respect, viz., the length of the translator itself, the approximately 5,000 half-words occupied by the translator (i.e., just about 1/6 of all the external storage available) are not a real limitation imposed on programs to be run on the computer, since it very seldom happens that a long program, or programs using voluminous data, are written in SAKO and executed in translated version at once. Most commonly such "long" programs (if written in SAKO) are first translated and printed by the computer in machine code or SAS-W, modified or adjusted "manually" (if necessary), and only then fed into the computer a second time for production runs; and now the full SAKO-translator is not necessary.

Bibliographical notes. A short but thorough description of the SAKO system is to be found in [*44*] and [*49*], both papers written in English. At the Warsaw Conference on Automatic Programming, held in 1961, several other papers [*2, 62, 63*] concerning SAKO were presented, and a limited number of copies of the English version of these is still available on request from IMM. In addition, the IMM has published two Polish reports [*17*] and [*61*] about the implementation of external programming languages on the XYZ and ZAM-2 computers. Arithmetic formula translation is described in [*2*], and the use of subroutines is explained (briefly) in [*49*] and in full length in [*63*]. Assembly routines for SAKO translation are described in [*62*]. Finally, for the fullest description of the SAKO system, [*45*] should be consulted.

2.3 KLIPA

2.3.1 The Pragmatic Approach

The Computation Center of the Polish Academy of Sciences originated from the acute demand for large volume computation, posed by the Institute of Nuclear Research of PAS and other divisions of the Academy. The CC was set up on the basis of a URAL-2 computer, which at the time of its installation was the biggest and the fastest one in Poland. Like most of the Soviet-made computers, the URAL-2 does not possess any facilities for alphabetic input, the only standard input being the octally punched film tape, operated on the closed-loop principle reminiscent of the external memory used for some of the early IBM computers.[16] When reading this and the following section, the reader should bear in mind two facts: The programming staff of the CC was trained for coding in the internal (octal) language of the URAL-2 computer; and from the first days of its existence, the CC was overloaded with a continuous flow of orders for computations that had to be carried out.

These facts are familiar to many Computing Laboratories, both East and West, and thus the programming policy followed by the CC is, perhaps, a typical one in such a situation. It starts from two principal premises: that programming in internal language is slow and apt to cause many hard-to-discover errors, and that any automatic programming system requires many long preparatory steps and that the

[16]In order to permit input of alpha-numeric characters, a paper tape reader was attached to the computer at CC.

translation from programs written in an autocode is likely to upset tightly tailored production schedules. Therefore, a pragmatic decision was made: to construct a coding system which would reduce the most mechanical portions of the programmer's work and would not increase substantially the input time for programs.

The coding system worked out at the CC, the Klipa, is the practical outcome of this decision [*22, 23*].

As far as the single instruction is concerned, Klipa does not change the appearance of the operational part of an instruction. It is preserved in machine language, i.e., consists of two octal digits. There seemed to be no important reason for changing this form of the operational part, especially since all the programmers knew these forms by heart. The address part, on the other hand assumes the fully symbolic form of subscripted variables. The only limitation imposed is that the subscript, enclosed in parentheses, may be a linear function of one variable only. Hence, r, kappa (5x + 73), and april (23) are correct addresses, while pi (sigma + rho) is not.

A Klipa program is divided into sections of arbitrary length. The interchange of sections, i.e. the replacement of a section just executed by another, currently located on the drum, is done automatically, with the substantial additional feature that in this section interchange, as well as in other transfers between different levels of storage, special "programmed checks" are provided to eliminate possible errors of the type usually discovered by the parity checking (which is absent in the Ural-2 computer).

Another function of Klipa is the assembling of a section from separate pieces; this is done during the input of a program. Standard functions may be either added to the section body permanently, or called in, through a buffer area of core memory, any time the function is called for. This alternative way of using standard functions enables the programmer to follow successfully an optimization policy with respect to economy of locations and speed of calculations. Calling for the standard functions is accomplished by a simple jump instruction (the operational part of this instruction is 22) as in the following examples:

```
42 x      42 x
22 sin    22 rq
56 y      22 sin
          56 y
```

which are equivalent to the Algol statements:

$$y := \sin(x); \qquad y := \sin(\text{sqrt}(x)).$$

Special provision is made for the use of library subroutines, which may be either included with one of the sections of the program or stored on separate sections of tape.

Labeling in KLIPA is done by preceding the instructions to be labeled by a symbolic label, e.g.:

:kappa) 02 x.

Labels of the form: ⟨label⟩ + ⟨octal integer⟩) are acceptable and have obvious meaning.

All identifiers, i.e. variable names and labels, should be declared; this means that numerical (octal) values of the identifiers should be given in the form: x := 20, kappa := 7032. This information is used by the translator to replace symbolic names by their numerical values during the translation process. Hence, with the declarations given above, the instruction 42 kappa (x + 2) will be translated as 42 7054. There is a possibility of chain declarations, like:

$$\begin{aligned} \mathrm{x} &:= 5 \\ \mathrm{kappa} &:= 4\mathrm{x} + 3 \\ \mathrm{june} &:= \mathrm{kappa}\ (12\mathrm{x} + 5). \end{aligned}$$

Labels should be declared in a similar manner, with two exceptions:

(i) Labels for indicating points to which control is to be transferred may take the form of primed letters: a', b', . . . , z', and in such cases no declaration is needed.

(ii) There is a special label "pocz" (short for a Polish word meaning "beginning") which is automatically declared as the smallest possible label in the sections of the given program; thus all labels may be made relative to this particular one by means of a sequence of chain declarations, and hence any absolute addressing may be avoided altogether.

This system of labeling and declaration may look a little peculiar, and a few words are needed to explain why there is no fully automatic storage allocation or labeling in KLIPA. The main reason is that KLIPA was devised to facilitate the work of skilled programmers who use in their everyday routine work methods of absolute addressing and know how to take full advantage of these methods. Thus, any automatic storage allocation would appear to them artificial, and perhaps wasteful. Now the KLIPA labeling and declaring system gives to a programmer all the possibilities which the absolute addressing could have given, and in addition it simplifies to a large extent the actual work to be done. By preserving the freedom of applying all the tricks programmers knew, KLIPA won their confidence, while it is well known that persons used to programming in machine language usually mistrust all forms of automatic programming. On the other hand, the special label "pocz" makes it possible to introduce to a newcomer a completely relative-addressed programming system, thus avoiding much of the laborious task of checking and changing absolute addresses.

While single instructions in KLIPA are simple and do not differ very much from machine language instructions, a sophisticated set of red-tape subroutines for KLIPA was devised to facilitate the programmer's work as much as possible. Moreover these unspectacular routines were largely written with future ALGOL implementation in mind. It is hoped that most of the existing red-tape operations will be included into an ALGOL compiler without many changes.

Translation of the KLIPA programs is performed on input, and with a speed limited only by the paper tape reader capability to read 400 characters per second. There is no necessity of reading in an entire line, followed by decoding, identifying, and storing of the recoded information. All these stages of translation are performed by the KLIPA translator as soon as a single character is read in and, as a rule, the translator processes (in the sense that a translator is supposed to) the information carried by a character before the next one can be sensed. On the average it is felt that only if tape readers were faster than 1000 characters/second, the translation would be limited by the programming.

This relatively high speed of translation is due to two basic principles incorporated in the KLIPA: (i) use of the characteristic function method for identification of symbolic names, and (ii) the machine instruction 30 α, which permits quick realization of multiway switches and which has been widely used. Since the former reason is discussed in the following section, we shall now devote our attention to the latter.

The instruction 30 α, where α denotes any octal integer not exceeding 7777, is obeyed in the following manner: the sequence of two instructions

$$\begin{gathered} 30\ \alpha \\ I_0 \end{gathered}$$

results in executing instruction I_1 which is composed of I_0 and of the content of the register whose address is α. The composition of I_1 is described by the formula $I_1 = I_0 + (\alpha)$, where (α) denotes content of register α, and both (α) and I_0 are treated as octal numbers (including the two most significant digits forming the operational parts). Hence, e.g., if $\alpha = 7000$ and $(7000) = 00\ 0002$, the sequence

```
30 7000
42 0030
```

is equivalent to 42 0032.

A multiway switch in the translator may then be arranged as follows. Let the register α always be loaded with the character read in most recently. Then the sequence

```
30   α
22   k,
```

where 22 is the unconditional jump instruction, results in a jump to either of 32 registers following the kth (32 is the number of different binary representations of characters available on the 5-channel paper tape). It is easy to see how this may be used for checking the formal correctness of the programs and for interpretation of various symbols appearing in the program.

2.3.2 The Characteristic Function Method

In many problems of non-numerical data processing, such as construction and use of compilers, translators and other types of autocodes, business data processing, mechanical translation, etc., there arises the problem of identification of alphabetic inscriptions coded by means of paper tape (or card) punching devices. Quite often the only identification needed consists in assigning to each inscription a particular integer, which may be thought of as the location address where the relevant numerical information is stored. For example, in autocode practice the information stored may be the address of the memory cell assigned to a given alphabetic variable, or, in some cases, just the current value of the variable itself. In natural language translation, the numerical information stored in the location whose address is associated (by the identification procedure) with a given inscription (i.e., with a particular word of the text to be translated) would be, perhaps, composite; it would contain coded information about the grammatical structure and properties of the inscription (e.g., tense, mode, and aspect of verb), a "signal" bit indicating whether the word is unambiguous (in the sense of not having homonyms) and the address of the location of the corresponding word of the language into which the translation is carried.

These examples explain what is meant by "identification" in the following part of the section. It is perhaps worthwhile to mention that the identification procedure suggested is especially powerful when combined with a method analogous to the one described at the end of Section 2.3.1.

For the sake of simplicity, we confine ourselves to inscriptions coded by means of standard teleprinter equipment operating on five channel tape. Thus any inscription is a sequence of rows of holes, the number of holes in a row varying from zero to five. Each row of holes may be thought of as a binary representation of an integer in the range 0 through 31. On the other hand, these rows may be considered as digits in 32-basis-notation system. Hence, each complete inscription may be interpreted as a number recorded in this system. Converting these numbers into the familiar decimal (or binary) notation system we would obtain a *natural identification rule*: Each inscription is associated with a store location whose address is given by the decimal (or binary) interpretation of the teleprinter-coded representation of the inscription. For example, the inscription "us," coded by a teleprinter working in the Second International Teleprinter Code as

```
oo o. .
o. o. .
```

(where punched holes are denoted by circles, and nonpunched holes are represented by dots) would be associated with the $28 \times 32 + 20 = 916$th store location, and the inscription "sin," coded as

```
o. o..
.o o..
.. oo.
```

would be associated with store location $20 \times 32^2 + 12 \times 32 + 6 = 20{,}870$. However, this natural interpretation rule would give rise to excessive memory requirements, since for identification of inscriptions belonging to a vocabulary consisting of at most j-letter words a storage of 32^j locations would be needed. To avoid this, two simple methods may be used. The first reduces the number of letters in an inscription used for identification purposes, which decreases j. The second consists in a kind of contracting procedure applied to the results of the natural rule.

In natural language translation the first method is unacceptable since cutting off part of a word may lead to considerable misunderstanding. In automatic programming both methods are useful, and in fact both were applied in KLIPA.

Before considering in some detail a variation of the second method, called *characteristic function method*, let us imagine that we have applied the natural rule to a vocabulary of the following structure:

(i) There are no homonyms in the vocabulary. (This is not a substantial restriction and may easily be overcome.)

(ii) The words in the vocabulary have at most j letters.

(iii) Words are divided into groups in accordance with their length.

Thus, there would be j groups of words, G_i, $i = 1, 2, \ldots, j$ of n_i elements (inscriptions) each, the total $\sum n_i$ being the number of inscriptions in the vocabulary. The application of the natural rule would result in plotting the inscriptions along a ray. Inscriptions of the G_i group would belong to the segment $\sigma_{i-1}\sigma_i$, where $\sigma_i = 32^i$.

Now, let us define for all integers from 0 to 32^j a function

$$\begin{aligned} \rho(\xi) &= 1 \text{ if there is an inscription associated with this integer } \xi \\ &= 0 \text{ otherwise.} \end{aligned} \qquad (2.3.2.1)$$

Introducing a continuous variable x, we define

$$\rho(x) = \rho(\xi) \qquad \text{for} \qquad \xi - \tfrac{1}{2} \leqslant x < \xi + \tfrac{1}{2}. \qquad (2.3.2.2)$$

It is easy to see that

$$\bar{\rho}_i = \frac{\int\limits_{\sigma_{i-1}}^{\sigma_i} \rho(x)\,dx}{\sigma_i - \sigma_{i-1}} \leqslant 1, \qquad \text{for } i = 1, 2, \ldots, j. \qquad (2.3.2.3)$$

The $\bar{\rho}_i$ may be considered as a measure of filling the memory with vocabulary items, and the difference $1 - \rho_i$ measures the relative number of wasted registers.

Our aim is to find a transformation of the variable x, $y = \Phi(x)$, which satisfies the following conditions:

(i) If x is an integer, y should be in integer too

(ii) If $x_1 \neq x_2$, $y_1 = \Phi(x_1) \neq \Phi(x_2) = y_2$

(iii) $\sum\limits_{i=1}^{j} (1 - \bar{\rho}_i(y)) = \min$, where

$$\bar{\rho}_i(y) = \rho_i(\Phi(x)) = \frac{\int\limits_{\Phi(\sigma_{i-1})}^{\Phi(\sigma_i)} \rho(\Phi(x))\, d\,\Phi(x)}{\Phi(\sigma_i) - \Phi(\sigma_{i-1})} \qquad (2.3.2.4)$$

(iv) The function Φ should be easily performable by computers. It has been found experimentally that a linear transformation of the form $y = a_i x + c_i$, with coefficients a_i, c_i suitably chosen for each G_i, may be used to advantage.

On considering the choice of the coefficients two important cases are to be distinguished.

Let all the integers ξ, $\sigma_{i-1} \leqslant \xi < \sigma_i$, such that $\rho(\xi) = 1$, be subscripted in ascending order $\xi_1, \xi_2, \ldots, \xi_{n_i}$. Let μ denote the greatest common divisor of all positive integers δ_r, defined by

$$\delta_r = \xi_{r+1} - \xi_r, \qquad r = 1, 2, \ldots, n_i - 1.$$

The cases to be distinguished are (i) $\mu > 1$, and (ii) $\mu = 1$, i.e., the case of mutually prime numbers δ_r.

In the case (i) we define $a_i = 1/\mu$ and, by the "*fixed point* division" transformation $y = x/\mu$, we reduce the case to (ii) and considerably increase ρ_i as defined by (2.3.2.4.)

In the case of mutually prime numbers δ_r we do not apply any multiplication, i.e. $a_i = 1$, but we try to pick a value c_i which shifts the segment $\sigma_{i-1}\, \sigma_i$ left so that occupied positions of the segment coincide with empty positions of preceding segments. It may happen that we are unable to find enough "free" spaces to reallocate all the occupied locations of the segment $\sigma_{i-1}\sigma_i$; nevertheless, in practice it is nearly always possible to shift the segment left by a number of locations. Furthermore sometimes it is possible to find a quasi-common divisor, i.e., a number μ^* such that for all $\xi_i \in (\sigma_{i-1}, \sigma_i)$ we have $\xi_i/\mu^* = p_i + \varepsilon$, where p_i is an integer and ε is the machine-representable unrounded remainder. Then, a transformation $y = x/\mu^* - \varepsilon$ will be very suitable for reducing memory requirements.

The transformations described here may easily and rapidly be performed by a computer on any inscription belonging to the vocabulary for which the coefficients have been computed, i.e., the method is applicable to *fixed* dictionaries only, since addition of a new word (inscription) brings about a re-evaluation of all coefficients. Another restriction is imposed by the necessity of the inscription being read in completely before the identification can start.

Another variation of the method, viz., that of characteristic functions, allows for the identification procedure to be carried out simultaneously with reading in of the inscriptions. This variation has been employed for KLIPA translations.

Consider the first n letters of a given inscription. Letters $n + 1$, $n + 2, \ldots$ are disregarded. If an inscription contains fewer than n letters, the following procedure is interrupted as soon as the last letter is sensed. As soon as the jth letter of the inscription (treated as jth digit of the "base 32" number corresponding to the inscription) is being read in, the computer evaluates:

```
$f_j^* := a_j f_{j-1} + x_j + c_j$;
**if** $f_j^* \geqslant M$ **then** $f_j := f_j^* - K$ **else** $f_j := f_j^*$;
```

For a set of inscriptions used by the KLIPA language the following experimentally chosen values of constants have been found to satisfy the minimization demand imposed on (2.3.2.4):

$n = 4,\quad M = 255,\quad K = 179,\quad a_i = 2$ for $i = 2, 3, 4,\quad c_1 = 0,$
$c_2 = 20,\quad c_3 = -58,\quad c_4 = -25,\quad f_0 \equiv 0.$

Using this transformation it became possible to reduce the number of wasted registers to 20 per cent. For the remaining waste we are amply rewarded by the impressive speed of identification of the inscriptions that are used in the KLIPA programs.

Bibliographical notes. A thorough formalized description of the KLIPA language is given in [*22*]. Some, perhaps the most interesting, features of the KLIPA translator are to be found in [*23*]. Paper [*20*] explains the symbolic addressing method for the EMAL-2 computer, and in a sense may be considered as the source of the methods used in the KLIPA language. General considerations on relations between programming and computer structure are contained in [*23*] and developed in [*21*]. Papers [*65, 66*] include, among others, an exposition of KLIPA group views on the value of automatic coding for business and scientific applications of computers.

3. Other Countries of Eastern Europe

In the present section we shall give the reader the opportunity to get acquainted with a handful of results obtained in various countries of Eastern Europe. The variety of subjects to be mentioned in this section makes it impossible to give a consistent introduction to the section; thus we present the following three subsections as separate expositions of fine pieces of work.

3.1 Klouček-Vlček Symbolic Language for Economic Data Processing

It is a well-known fact that economic[17] data processing is greatly handicapped by lack of a suitable external language of programming. Attempts undertaken in Western Europe and USA have resulted in languages like COBOL or NEBULA, which (thanks to their inherent speech-like characteristics) are of considerable merit, especially for nonmathematically minded people. They do not, however, provide the rigorous notation needed for mathematical development in this field.

Two Czechoslovak authors, J. Klouček and J. Vlček, have proposed [*33*] a symbolic notation with which mathematicians working in the field of data processing could readily familiarize themselves, and which at the same time may be comparatively easy to learn for economists, accountants etc., though not as easily as, say, COBOL. The K-V notation is, to a certain extent, similar to that of formal logic and operator

[17]For brevity we use the words "economic data" to represent data relating to the national economy, business operation, and others having similar structure.

calculus. From the automatic programming point of view, the K-V system is closely related to the symbolic operators method of programming (cf. Section 1.1) and construction of the corresponding PP's should be fairly easy; however, no hardware representation of the K-V system is known to exist.

The K-V notation system, or K-V symbolic language, KVL, is particularly applicable to all problems of selecting, combining, and rearranging of economic information items and sets of such items. In other words, KVL is a language for description of data preparation, i.e., procedures that precede the actual calculations.

First of all, let us define the various forms of economic information: *Elementary economic information* is an item of information that cannot be further subdivided without losing significance. Examples of elementary economic information are number of employees, number of hours worked, etc. The elementary economic information has no inherent meaning and may be related to many other items of elementary information, e.g., number of hours of work a particular employee, or number of hours of work per month for a given factory.

Two or more items of elementary economic information related to each other and reflecting a concrete economic phenomenon are said to form a *compound economic information* item; e.g., the compound economic information consisting of the elementary economic information items: number of employees, hours of work, and total production output, corresponds to an economic phenomenon: number of hours spent by an average employee on one unit of production.

Finally, an *information assembly* is a set of compound information items.

The concepts of elementary information item, compound information item, and information assembly correspond to what are sometimes called item, record, and file, respectively. We shall use small letters to denote elementary economic information items, bold face small letters to denote compound information items, and bold face capitals to denote assemblies.

We define the following obvious relations between two pieces of elementary economic information a and b: $a > b$, $a < b$, $a = b$.

Symbols denoting compound economic information may be subscripted in order to show a particular arrangement (and number) of compound information items into an information assembly; e.g., $\mathbf{q}_i$ for $i = 1, 2, \ldots, 1000$ signifies that the economic phenomenon described by the compound information $\mathbf{q}_i$ occurs 1000 times, and the information assembly thus defined comprises 1000 cases.

Operations performable on economic information resemble operations with logical classes and may be denoted by the same symbols, with necessary modification of meaning of the symbols, which follows from the dynamic nature of the relations: they do not reflect established facts, but become true by virtue of the operations performed in accordance with the relation symbols used.

The fundamental operations performable on economic information are:

(i) *Comparison*, performable on two items of economic information of the same form, is denoted by $\overset{?}{=}$, e.g., $a \overset{?}{=} b$.

(ii) *Forming* a higher form of economic information from lower ones is associated with the symbol $\in$, e.g. $\mathbf{q}_i \in \mathbf{A}$ means that the assembly **A** is formed from the compound information items $\mathbf{q}_i$.

(iii) *Extracting* a lower form of economic information from a higher one is denoted by $\cap$, e.g., $\mathbf{q}_i \cap \mathbf{A}$ means: the compound information $\mathbf{q}_i$ is extracted from the assembly **A**.

The derived operations performable on economic information are either sequences of the operations defined above, or belong to one of the following:

(iv) *Marshalling* of an assembly according to the elementary information contained in compound information that form the assembly is denoted by $j(\mathbf{A})$, where j is a particular elementary information. The compound information is marshalled within the assembly in accordance to specified, usually numerical, rules.

(v) *Ordering* is a special kind of marshalling and is used to arrange the compound information forming an assembly in ascending or descending order with regard to the numerical values of certain elementary information contained in the compound one. This is written symbolically in KVL as $\mu_{+}j(\mathbf{A})$ or $\mu_{-}j\,(\mathbf{A})$, depending on the ascending or descending order desired. The plus sign as subscript may be omitted.

(vi) *Joining* two or more items of information of the same form into a new item of information of the same form as the original ones; the associated symbol is $\cup$. This operation is not performable on elementary information.

(vii) *Inclusion* of compound information into an assembly is denoted by $\subset$, e.g., $\mathbf{q}_i \subset \mathbf{A}$. There is an opposite operation, called

(viii) *Exclusion*: $\mathbf{A} \supset \mathbf{q}_i$.

In addition to the above, the KVL uses the following symbols:

[**A**]	Denotes an ordered set of elementary economic information items.
$\overline{\mathbf{A}}$	Denotes the complement of the assembly **A**.

$\Rightarrow$	Is used as "short-hand sign" for the words: "generates," "results," etc.
$\{\}$	Curled brackets are used in order to explain the preceding operations in more detail.

The symbols $=$, $+$, $-$, ., : are used in their general sense.

Now let us consider a simple example. We shall write a KVL program for processing the assembly **A** formed by n items of compound information $\mathbf{q}_i$ $(i = 1, 2, \ldots, n)$. We would like to obtain as output the following data: (1) the compound information that contains the smallest elementary information b, (2) the greatest elementary information c from each of the subassemblies created by marshalling **A** according to a, and (3) the sum of all items of elementary information c contained in these subassemblies. The sought-after quantities may be denoted in KVL as: $\mathbf{q}\,\{b = \min\}$, $\max c\,\{\,a\,\}$, $\sum c\,\{\,a\,\}$; $\mathbf{q}_i = (a, b, c)$.

The problem is solved in the following manner:

(1) $\mathbf{q}_i \in \mathbf{A}$;	
(2) $a\,(\mathbf{A}) \Rightarrow [\mathbf{A}]$;	Marshalling according to a.
(3) $[\mathbf{A}] \supset \mathbf{M}_j\, j = \min a, \ldots, \max a$;	The ordered assembly **A** is divided into subassemblies $\mathbf{M}_j$.
(4) $\mu_{+}\, b(\mathbf{M}_j) \Rightarrow [\mathbf{M}_j]$; $\{\mathbf{q}_k \in [\mathbf{M}], \mathbf{q}_{k.j} \in [\mathbf{M}_j]\}$	Subassemblies are ordered with respect to b.
(5) $b \cap \mathbf{q}_{1.j} = b_{1.j}$;	The first (smallest) item of information b is extracted from all the subassemblies.
(6) $b_{1.j} \overset{?}{=} \min b_{1.j}$;	From the extracted items the smallest b is determined.
(7) $\mathbf{q}_{1.j} \cap [\mathbf{A}], \{b_{1.j} = \min\}$;	The compound information **q** with the smallest b is extracted from [**A**].
(8) $c \cap \mathbf{q}_{k.j} = c_{k.j}$;	Item c is extracted from all the compound information **q** which belong to all **M**.
(9) $j(c_k \in \mathbf{C}) \Rightarrow [\mathbf{C}_j]$;	This information is ordered into independent subassemblies (in which "compound information" consists of single elementary items!).
(10) $\mu_{-}(\mathbf{C}_j) = [\mathbf{C}_j]$;	

(11) $c_1 \cap [\mathbf{C}_j] \overset{?}{=} \max c_j$;	The first (greatest) element is extracted from each of the ordered subassemblies C, and the greatest of the items thus extracted is found.
(12) $\sum_{k=1}^{x} c_{k.j} = f_j$;	The sum of the extracted items c in each subassembly is evaluated.

KVL, as may be seen from the above, does not conform to popular demands for a business data processing language generally understandable on the executive or managerial level. Moreover, even from a purely mathematical point of view some improvements, giving more flexibility, are urgently needed. Nevertheless, KVL should be regarded as a first successful step towards a formalized external programming language for economic data processing. It may be hoped that discussion and cooperative effort of interested persons will help to build a better language, which will remove many of the difficulties commonly encountered in this field.

3.2 Sedlák's Program for Investigation of Solutions of Differential Equations

There are numerous methods for the numerical treatment of ordinary differential equations, and perhaps no universal rule for an *a priori* decision as to which of them is to be used for a particular set of equations. On the other hand the decision-making process in this case is rather simple when only its logic is considered, and becomes lengthy and tedious only when all calculations necessarily involved are taken into account. Thus, naturally enough, it is desirable to program this procedure for a digital computer in order to enable the mathematician operating the computer to arrive at correct decisions quickly and safely. Such a program, supplied with some auxiliary subroutines not directly connected with the problem of finding the best numerical method, has been developed by J. Sedlák [*52*] and used by the staff of the Institute for Mathematical Machines in Prague on the SAPO digital computer. It will become clear from the remainder of this section that Sedlák's program is practically machine-independent and thus may be used on other types of computers.

The program is used in order to investigate the solution of the set of differential equations

$$y_r'(x) = f_r[x, y_1(x), y_2(x), \ldots, y_s(x)], \qquad r = 1, 2, \ldots, s \qquad (3.2.1)$$

with initial conditions

$$y_r(x_0) = y_{r0}, \qquad r = 1, 2, \ldots, s. \tag{3.2.2}$$

The solution of (3.2.1) is desired in the form of a matrix y_{ri}, $r = 1, 2, \ldots, s$, $i = 1, 2, \ldots, n$; in which the ith row corresponds to values of the unknown functions y_r assumed for the value x_i of the independent variable x; $x_i \in I_x$, where I_x is known as the range of integration. We shall divide the methods of numerical integration into two classes:

(i) Direct methods (step-by-step methods), i.e., those methods in which, in order to obtain y_{ri} ($r = 1, 2, \ldots, s$), information about only $y_{r,i-1}$ is used. The best known method of this class is that of Runge-Kutta[18]:

$$\begin{aligned}
y_{ri} &= y_{r,i-1} + \frac{h}{6}(k_{r1} + 2k_{r2} + 2k_{r3} + k_{r4}),\\
k_{r1} &= f_r(x_{i-1}, y_{1,i-1}, y_{2,i-1}, \ldots, y_{s,i-1}),\\
k_{r2} &= f_r(x_{i-1} + h/2, y_{1,i-1} + hk_{11}/2, \ldots, y_{s,i-1} + hk_{s1}/2),\\
k_{r3} &= f_r(x_{i-1} + h/2, y_{1,i-1} + hk_{12}/2, \ldots, y_{s,i-1} + hk_{s2}/2),\\
k_{r4} &= f_r(x_{i-1} + h, y_{1,i-1} + hk_{13}, \ldots, y_{s,i-1} + hk_{s3}).
\end{aligned} \tag{3.2.3}$$

(ii) Indirect methods, i.e., methods in which values of y_{rj} for

$$x_j \in I_x^k \equiv \{x_{i-k}, x_{i-k+1}, \ldots, x_{i-1}\} \subset I_x \tag{3.2.4}$$

are used in order to obtain values of y_{ri}. There are two subclasses of the indirect methods. Let us consider, e.g., a method due to Störmer and Adams:

$$y_{ri} = y_{r,i-1} + h\left(-\frac{3}{8}y'_{r,i-4} + \frac{37}{24}y'_{r,i-3} - \frac{59}{24}y'_{r,i-2} + \frac{55}{24}y'_{r,i-1}\right), \tag{3.2.5}$$

$$y_{ri} = y_{r,i-1} + h\left(\frac{1}{24}y'_{r,i-3} - \frac{5}{24}y'_{r,i-2} + \frac{19}{24}y'_{r,i-1} + \frac{9}{24}y'_{ri}\right). \tag{3.2.6}$$

Formula (3.2.5) defines the S-A extrapolation method, and (3.2.6) the S-A interpolation method.

Sedlák's original program made provisions for methods (3.2.3), (3.2.5), (3.2.6) and some additional ones. Explicit knowledge of methods that may be applied to a given set of equations (3.2.1) is not necessary, since it is supposed that the computer is supplied with all the subroutines necessary to perform calculations according to the pertinent scheme.

[18]Formula (3.2.3) gives the method in its classical version; in contemporary computation practice Gill's modification of the process is generally preferred.

The programmer has to prepare input data consisting of one code word **OI** representing in a symbolic manner the desired order of integration methods, and of several code words $\mathbf{g}_i$ specifying in symbolic manner the desired output quantities.
The code word **OI** assumes the form

$$\boxed{\mathrm{m} \mid \mathrm{z} \mid \mathrm{k} \mid \mathrm{s} \mid \mathrm{v}}\,, \tag{3.2.7}$$

consistent with the computer word structure in SAPO. All letters in (3.2.7) are to be replaced by binary numbers with the following meaning:

m	Number of steps to be integrated by the indirect method
z	Symbolic number denoting the type of the indirect method
k	Number of steps to be integrated by the direct method
s	Number of equations in (3.2.1)
v	Special variable: $v = 1$ when the integration step length h is predetermined by the programmer, $v = 0$ if h is to be chosen by the computer itself.

The code words $\mathbf{g}_j$ have the form

$$\boxed{t_j - t_{j-1} \mid d_j \mid p_j} \tag{3.2.8}$$

In order to explain the meaning of (3.2.8) we consider the integration procedure as divided into j stages, each stage differing from the others with respect to the desired output data; e.g., for the first five steps we would like to have all y_{ri} printed, for the next ten steps only $y_{2\nu,i}$, and so on. Now, let t_j denote the number of the last step of integration belonging to the jth stage. A finite sequence of integers $t_j - t_{j-1}$ consisting of as many numbers as there are different stages may serve a dual purpose: first, it determines the length of each stage; second, when the words $\mathbf{g}_j$ are located consecutively in machine memory, these integers determine *which* stage is being executed at any given time. The binary integer d_j in (3.2.8) is a symbol indicating which particular quantities are to be printed out during the jth stage. In Sedlák's original program, $d_j = 2$ calls for output of y_{ri} ; $d_j = 4$ calls for $\triangle y_{ri}$, $d_j = 6$ causes printing of y_{ri} and y'_{ri}, and so forth.

The integers p_j in (3.2.8) denote the over-all number of quantities to be printed during the jth stage.

In addition to the code words $\mathbf{g}_i$ and **OI**, the programmer has to prepare a subroutine for evaluating the right-hand expressions of (3.2.1) and specify either a fixed h or a tentative initial h and the desired accuracy ε. The initial values (3.2.2) form the last part of the input data.

All the input is to be recorded in fixed store locations, except, of course, the subroutine for the evaluation of the right-hand expressions, which has fixed beginning only.

Sedlák's program includes a special subroutine for changing and choosing the step length h in such a manner that it will satisfy the desired accuracy requirement given by ε. Since the subroutine is based on a "double-the-h-repeat-calculations-and-compare-results" process, it is essentially independent of the integration method used and thus may be called in from any arbitrary point of the program, as produced by the compiler.

For a student of automatic programming the interesting part of Sedlák's program is an assembly routine, which on the basis of the sequence of $\mathbf{g}_j$ and **OI** words prepares the program. For brevity's sake, we cannot give many details of this routine. The interested reader is advised to consult the original paper [*52*], where complete flowcharts are included. We shall only point out some remarkable features of the Sedlák program which make it very convenient for the purpose stated at the beginning of the present section.

First of all, the program is self-restoring. This means that, once the equations (3.2.1) are specified by input of right-hand-subroutines, any number of different **OI** and $\mathbf{g}_j$ may be executed without additional changes. In other words, the data tape may consist of many blocks, say, $\mathbf{OI}_1\{\mathbf{g}_j^1\}$, $\mathbf{OI}_2\{\mathbf{g}_j^2\}$, . . . , thus permitting comparison of accuracy obtained and time consumed by various methods and combinations of methods.

Another interesting and important feature of the program is that it is "equation independent." This means that, once the program is fed into the computer, it may be used to investigate many different sets of (3.2.1). This is achieved by changing not only **OI** and $\{\mathbf{g}_j\}$, but also the subroutines for the right-hand expressions.

Finally, let us observe that Sedlák's program is equally handy for investigation and for actual calculations, hence it is possible to proceed to integration over long range immediately after the method is experimentally chosen.

Concluding this section we point out that Sedlák's program may be compared with the work of Stogniĭ [*58*], mentioned briefly in Section 1.5, and seems to originate from the same theoretical trend, viz. ideas supported by Glushkov's (Kiev) school of automatic programming.

3.3 Kalmár's Detachment Procedure

In this section we discuss an exceptionally interesting procedure due to the Nestor of Hungarian computer people, László Kalmár of Szeged.

The procedure was devised in order to simplify translation of arithmetic formulas into M-3 computer code. M-3 is a two-address computer, with a peculiar operation list which for a given arithmetical operation Θ presents four different machine instructions:

$$\begin{array}{llll l} 0 & \Theta & a & b & b\ \Theta\ a \rightarrow r, b; \\ 1 & \Theta & a & b & b\ \Theta\ a \rightarrow r; \\ 2 & \Theta & a & b & r\ \Theta\ a \rightarrow r, b; \\ 3 & \Theta & a & b & r\ \Theta\ a \rightarrow r; \end{array} \tag{3.3.1}$$

In (3.3.1) a and b are addresses of operands, r denotes a special register of the arithmetical unit, and $\rightarrow$ means loading the registers indicated on the right-hand side with the result of the operation stated on the left. Thus, e.g., $b\ \Theta\ a \rightarrow r, b$ means to load *both* r and b with the result of the operation $b\ \Theta\ a$. Not all arithmetic operations are available in all four varieties (3.3.1), but by using the unconditional jump instruction $2\,J\ a\ b$ with meaning: "$r \rightarrow r, b$; jump to a", we may form pairs like

$$\begin{array}{llll} 1 & \Theta & a & b \\ 2 & J & * + 1 & b, \end{array} \tag{3.3.2}$$

where $*$ denotes the address of the instruction in which it occurs. Since the operation 1 Θ is available for all Θ, the pairs (3.3.2) may be used as a substitute for 0 Θ if this should be lacking for a given Θ; and similarly for the other forms of single instructions appearing in (3.3.1). Hence, for the sake of brevity, we shall consider all four forms (3.3.1) as available for all arithmetical operations Θ.

The feature of the M-3 two-address computer described by (3.3.1) may be advantageously used for automatic programming purposes, namely, for optimizing subroutines of the translator: the result of any operation need not be stored, if it can be used as first operand for the next instruction. (It is worth noticing that here we have an implicit answer to the question whether one-, two-, or three-address computers are best suited for automatic programming).

The foregoing implies that the *chain operator*

$$(\ldots((a_0\ \Theta_1\ a_1)\ \Theta_2\ a_2)\ldots)\ \Theta_\nu a_\nu \rightarrow b \tag{3.3.3}$$

rather than an operator of the form $a\ \Theta\ b \rightarrow r$ should be considered as the simplest form of nontrivial arithmetical operator (assignment statement). Indeed, the chain operator (3.3.3) may be readily translated into M-3 machine language:

$$\begin{array}{llll}
1 & \Theta_1 & a_1 & a_0 \\
3 & \Theta_2 & a_2 & \\
3 & \Theta_3 & a_3 & \\
 & \dots & & \\
3 & \Theta_{\nu-1} & a_{\nu-1} & \\
2 & \Theta_\nu & a_\nu & b,
\end{array}$$

where the omitted addresses are immaterial.

Thus, the main objective for research on automatic programming for M-3 and allied computers is to find an algorithm for decomposition of any arithmetic operator into a finite number of chain operators. The problem has obviously a trivial solution, obtainable with the help of algorithms constructed for a three-address computer; but then, by virtue of all chain operators being reduced to simplest three-address operators, $a \ \Theta \ b \rightarrow c$, there will be many unnecessary transfers to and from machine memory. Hence, more correctly stated, the objective is to find a decomposition algorithm which minimizes the number of chain operators needed to represent the given arithmetical operator. Besides, the algorithm should be "sufficiently simple," for too complicated an algorithm may easily consume a large part of the time gained by the optimization procedures.

Let us, following Kalmár's paper, consider arithmetic programs, i.e., finite sequences of arithmetical operators, each assuming the form $e \rightarrow v$, where e denotes an arithmetic expression and v denotes a variable. Arithmetic operators forming a program are separated by semicolons, and each of the arithmetical expressions is enclosed by at least one pair of parenthesis, except when it consists of a single variable, e.g.,

$$(a + b) \rightarrow x; \qquad (x\,(a - c)) \rightarrow a; \qquad a \rightarrow x \tag{3.3.4}$$

is an arithmetic program.

Expressions of the form

$$(v_1 \ \Theta \ v_2), \tag{3.3.5}$$

where v_1 and v_2 are variables, and Θ is an operation symbol, are called *first-order chain expressions.* Chain expressions of higher order are defined recursively, viz.,

$$(c \ \Theta \ v) \tag{3.3.6}$$

is a chain expression of ith order if c is a chain expression of $(i - 1)$st order. Chain expressions of any order will be called *chains.* An arithmetical operator of the form $c \rightarrow v$, where c is a chain, is called a *chain operator*, and an arithmetic program consisting of chain operators and operators of the form $v_1 \rightarrow v_2$ only is called a *chain program.*

Consider now the set $\mathbf{V}$ of all the variables pertinent to a given arithmetic program, and the set $\mathbf{V}'$ of the *essential variables* of the same program. We do not define essential variables in any specific manner; as a matter of fact in the following we only suppose that

$$\mathbf{V}' \subseteq \mathbf{V}. \tag{3.3.7}$$

Two arithmetic programs P_1 and P_2 are said to be *equivalent relative to the essential variables* $\mathbf{V}'$, which is denoted $P_1 \sim P_2\ (\mathbf{V}')$, if the numerical values attached to the essential variables by both programs, P_1 and P_2, are the same.

Two facts deserve to be mentioned at this stage:

(i) Kalmár's definition of the equivalence is much more precise than the intuitive explanation given here.

(ii) The reader is advised to bear in mind the fact that the set of all variables pertinent to a program may include some variables not given explicitly in the algorithm which is realized by the program—e.g., working cells should be considered as such variables. The same holds true for $\mathbf{V}'$.

At the beginning of the present section we have arrived at a conclusion concerning the objective of research on automatic programming for "M-3-like" computers.

This conclusion may now be formulated as follows:
To find a decomposition algorithm which, for a given arithmetic program P_1 and set of essential variables $\mathbf{V}'$, produces a chain program P_2 which is equivalent to P_1 relative to $\mathbf{V}'$.

Kalmár has proved a very important theorem, which describes formal conditions permitting application of procedures that lead to partially decomposed arithmetic programs and meet the demands of equivalence. Unfortunately, both the rigorous statement of the theorem and its proof are too long to be reproduced here, thus we restrict ourselves to an informal formulation of this theorem. Consider a program P and set of essential variables $\mathbf{V}'$. Let our objective be to "detach" an expression d occurring in P. This means that we are introducing into the program the operator $d \to w$, where w is a variable, and then replace (all, some, or none) occurrences of d by w. Variable w is called the *working variable of the detachment*. We divide our program into three parts P_1, P_2, and P_3, called head, trunk, and tail, respectively. This division depends on the choice of both d and w. Namely, the three following conditions must be obeyed:

(i) The expression to be detached must not change within the trunk.

(ii) The working variable of the detachment must not occur in the trunk, except for its occurences within the expressions to be detached.

(iii) The first (if any) occurence of the working variable in the tail has to be placed on the right-hand side of the symbol $\rightarrow$, and the working variable is allowed to be an essential variable if it occurs in the tail only.

Then the theorem says that if we detach d from the trunk, add the operator $d \rightarrow w$ as the last operator in the head, do not change the tail, and denote the new (decomposed) trunk by P'_2, we have the relation

$$P_1P_2P_3 \sim P_1P_2'P_3\,(\mathrm{V}'). \tag{3.3.8}$$

The theorem is, of course, entirely machine-independent and thus valid for any kind of addressing system used.

The detachment procedure proposed by Kalmár in the remaining part of his paper [*25*] is M-3 machine oriented, and hence we shall not make any detailed analysis of it. A few general remarks concerning the procedure may, however, clarify the underlying ideas. The expression d is chosen as a chain, and the primary goal of the procedure is to reduce the number of primitive symbols occurring in it, i.e. to shorten the program by detachment as much as possible, and at the same time produce as good conditions for the next detachment as possible; for the detachment procedure is, in a sense, recursive. A general rule for the best guess concerning the expression to be detached is to pick the first maximal chain (i.e., one which does not occur as a subexpression in the program), starting from the right. This simple rule is subject to many additions, since its straightforward application leads to clumsy compositions, as illustrated by the following example. Consider the program

$$\begin{gathered}(a + ((b + c)d)) \rightarrow u; \qquad ((b - c)/(b + c)) \rightarrow v; \\ (a - ((b + c)d)) \rightarrow w;\end{gathered} \tag{3.3.9}$$

The right most maximal chain is $(b + c)d$. Choosing t as a working variable we get

$$\begin{gathered}((b + c)d) \rightarrow t; \qquad (a + t) \rightarrow u; \qquad ((b - c)/(b + c)) \rightarrow v; \\ (a - t) \rightarrow w;\end{gathered} \tag{3.3.10}$$

Here, $(b + c)$ can still be detached, giving, with the help of a new working variable q, the following chain program:

$$\begin{gathered}((b + c)d) \rightarrow t; \qquad (a + t) \rightarrow u; \qquad (b + c) \rightarrow q; \qquad ((b - c)/q) \rightarrow v; \\ (a - t) \rightarrow w;\end{gathered} \tag{3.3.11}$$

whereas, detaching from (3.3.9) first $(b + c)$, we would get

$$(b + c) \rightarrow q; \quad (a + (qd)) \rightarrow u; \quad ((b - c)/q) \rightarrow v; \quad (a - (qd)) \rightarrow w; \tag{3.3.12}$$

from which (qd) should be detached, yielding

$$(b + c) \rightarrow q; \quad (qd) \rightarrow t; \quad (a + t) \rightarrow u; \quad ((b - c)/q) \rightarrow v; \quad (a - t) \rightarrow w; \tag{3.3.13}$$

Obviously (3.3.13) is a shorter, and thus "better," chain program than (3.3.11).

Note that no detachment procedure can give (3.3.13) from (3.3.10) since the first two terms of (3.3.10) form a chain, and thus are not subject to the detachment procedure.

From a purely pragmatic point of view Kalmár's algorithm has some minor disadvantages, pointed out by Kalmár himself. One of them is that the detachment algorithm tends to decompose even programs which are already machine acceptable. Another will be seen from the next example. In the case of the program $((a + b)c) \rightarrow d$; $(ef) \rightarrow g$; $(f - ((a + b)c)) \rightarrow h$; the algorithm will produce $((a + b)c) \rightarrow w$; $w \rightarrow d$; $(ef) \rightarrow g$; $(f - w) \rightarrow h$; introducing the unnecessary working variable w, and one unnecessary memory transfer, since the form $((a + b)c) \rightarrow d$; $(ef) \rightarrow g$; $(f - d) \rightarrow h$ would do equally well.

In spite of these and other minor drawbacks, the algorithm is a very convenient one and may well be adopted (with some modifications) for practical work.

4. Survey of Methods of Programming

In the preceding sections a more or less detailed analysis of the most important developments in the field of automatic programming has been presented. Now, a few words remain to be said about practical methods of programming employed in everyday work of computing establishments.

As far as is known from both published material and personal contacts with Soviet scientists, the prevailing programming system in the USSR is based on the two-step method, described in Section 1.1.[19] This method and its many variations imply a sharp division of routine work on programming between programmers who formulate problems in terms of one of the existing programming programs, and specify the

[19]For a discussion of programming methods for business applications, see Yu.I. Chernyak, Electronic Simulation of Planning Systems in USSR, *Data Processing* (*London*) **6**, No.1, 18–24 (1964).

functional meaning of separate blocks, and coders who carefully translate the symbols in which the program is written into those acceptable to the computer. Thus, with some oversimplification, we might have called the dominating Soviet programming system "the binary-coded automatic programming." In contrast to this are two important schools that of Shura-Bura and Ershov and that of Glushkov, both leading to a much higher degree of automation of programming; these should be considered as determining factors for future development. There is no doubt that when alphanumerical input devices become more widely available for Soviet-made computers, the tremendous theoretical work performed in that country will ripen into many interesting automatized programming systems.

In Poland, there is a positive tendency to introduce automatic programming for all computers that are operated in computational centers which perform calculations for various customers. In specialized computation centers, semiautomatic or even internal language programming is recognized as more efficient. Two languages are recommended for general use: a limited version of ALGOL and SAKO. Much research is devoted to the problem of enlarging SAKO so as to provide for business data processing; this is to be done by including many features and concepts of COBOL (the language so created is provisionally called SAKOBOL).

In other countries of Eastern Europe no distinct trend can be observed. In the German Democratic Republic [*26*] and in Czechoslovakia [*39*] programming in machine code, with some symbolic addressing, is at present the most commonly used programming technique. On the other hand, in Czechoslovakia a considerable amount of research is devoted to the preparation of an automatic programming system for the first Czechoslovak large-scale computer EPOS. Some traces of this work can be seen in Kindler's paper [*29*] (see Appendix 2). Simultaneously with that, ALGOL is becoming a widely accepted publication language: Many algorithms are published in ALGOL and subsequently translated into machine codes for testing purposes (e.g., [*30*]).

Thought-provoking remarks on automatic programming languages which are to be found in an extremely interesting paper by Culik [*8*] on languages generated by algorithms, and stimulating work by Svoboda on applications of Korobov's sequencing theory to addressing systems [*60*], show that, parallel with practical applications, some theoretical research on this subject is well advanced in Czechoslovakia.

In Hungary, most of the programming is done in machine language, though Kalmár's approach described in Section 3.3 above indicates that some successful attempts to automatize the programming process are being made there too.

In general, one may conclude that, in Eastern Europe, automatic programming is considered as a vitally important part of computer science, and a great deal of both labour and funds is devoted to theoretical and practical research in this field.

Appendix 1. Example of a Lyapunovian Program [41]

Let us consider a finite difference equation approximating Dirichlet's problem over a rectangular domain. Let the dimensions of the rectangle be $nh \times mh$, where n and m are positive integers, and h is an arbitrary positive number. The rectangle may be covered by a square net, each elementary square being of dimensions $h \times h$. Let the subscript j correspond to rows of the net, and the subscript i to columns, i.e., $i = 0, 1, 2, \ldots, n$; $j = 0, 1, 2, \ldots, m$. We shall denote by (i, j) a node of the net.

We are seeking the function f_{ij}, defined on the set of nodes (i, j), having prescribed values for all nodes $(0, j)$, (n, j) $(i, 0)$, (i, m), and satisfying at internal nodes the following equation:

$$f_{ij} = \tfrac{1}{4}\,(f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1}).$$

The solution will be found by an iterative process. For this purpose let f_{ij}^0 denote a first approximation. We define the operator A_{ij} as follows:

(i) The operator A_{ij} generates the quantity

$$f_{ij}^{k+1} = \tfrac{1}{4}(f_{i-1,j}^{k+1} + f_{i,j-1}^{k+1} + f_{i+1,j}^{k} + f_{i,j+1}^{k}),$$

(ii) transfers the result to the location previously occupied by f_{ij}^k,
(iii) generates $\rho_{ij}^k = |\, f_{ij}^{k+1} - f_{ij}^k |$,
(iv) selects the greater of the two numbers η, ρ_{ij}^k, and
(v) transfers it to the location previously occupied by η.

In other words, the operator A_{ij} may be represented by the scheme

$$[f_{ij}^*]\,[\rho_{ij} \rightarrow \rho]\; p(\rho > \eta)\,[\rho \rightarrow \eta]\,[f_{ij}^* \rightarrow f_{ij}],$$

where f_{ij}^* is the operator executing (i) and transferring the result to some fixed auxiliary location.

When the calculation has been performed for all internal nodes of the net, the final value of η is compared with a stored positive number ε. If $\eta > \varepsilon$ the computation is repeated, otherwise the computation is stopped (which is denoted by the operator OCT).

The computational scheme for this procedure is of the form:

$$\downarrow ([0 \rightarrow \eta] \prod_{j=1}^{m-1} \prod_{i=1}^{n-1} A_{ij})\, p\,(\varepsilon > \eta) \uparrow \text{OCT}.$$

The corresponding programming scheme is represented by

$$\overset{3}{\downarrow}[0 \rightarrow \eta]\overset{1,2}{\downarrow} A_{ij}F(i)\,p_j(i = n)\overset{1}{\uparrow} F^{-(n-1)}(i)F(j)p(j = m)\overset{2}{\uparrow}$$
$$F^{-(m-1)}(j)p(\varepsilon > \eta)\overset{3}{\uparrow}\text{OCT},$$

where the initial values of i and j are equal to one.
There are possible simplifications of the programming scheme, but the form given above may well be considered as typical of Lyapunov's method.

Appendix 2. Kindler's Algorithm for Programming of Arithmetic Formulas

A comparatively simple algorithm for programming arithmetic expressions for a three-address computer has been given by Kindler [*29*]. This algorithm can translate arithmetic expressions with variables, left and right parentheses, four basic binary operators, and the sign #, which is used to denote the terminal point of the expression. In this Appendix we give the algorithm in its simplest version. Some rather interesting features and possible extensions of this algorithm and complete proofs of its validity may be found in the original paper.

Let us adopt the following definitions:

(1) *variables* are *multipliers*,
(2) *multipliers* are *factors*,
(3) *factors* are *terms*,
(4) *term* + *factor* is a *term*,
(5) *term* − *factor* is a *term*,
(6) *factor* × *multiplier* is a *factor*,
(7) *factor*:*multiplier* is a *factor*,
(8) (*term*+*factor*) is a *multiplier*,
(9) (*term* − *factor*) is a *multiplier*,
(10) (*factor*×*multiplier*) is a *multiplier*,
(11) (*factor*:*multiplier*) is a *multiplier*,
(12) *term* # is an *arithmetic expression*.

If a string of characters (variables, parentheses, operators, and #) $a_1\, a_2 \ldots a_d$ is an arithmetic expression A, then the integer d is called the length of A.

The function p (priority) is defined on the set of all characters:

$$p(\times) = p(:) = 2$$
$$p(+) = p(-) = 1$$

otherwise $p = 0$.

Variables are divided into two kinds;

(i) actual variables A, B, . . . , which can be used in every arithmetic expression,
(ii) auxiliary variables $T1$, $T2$, . . . , which are used during the generation of a program.

The algorithm compiles the code for execution of expressions of length $d \geqslant 4$, and is written by means of an ALGOL-like language, with some obvious extensions included in the class of Boolean expressions. The procedure compile (k, V, $\circ$, W), where k is a nonnegative integer, V and W are actual variables, and $\circ$ is an operator, is meant to denote that:

(i) the generated code will have at least k instructions,
(ii) the kth instruction is $V \circ W \rightarrow Tk$.

A statement of the type $a[i]: = Tk$, means that the ith character of the expression is overwritten by the character Tk; q is the smallest index i of those a_i which are pertinent to the resulting program at the stage of compilation corresponding to a given value of k. In other words, characters $a_1, a_2, \ldots, a_{q-1}$ have been (at a given stage of compilation) already used, and thus are irrelevant for subsequent stages.

```
begin
$q := 1$; $k := 1$; $i := 1$;
A1: if $i \geqslant d - 1$ then go to A5;
    if $a[i]$ is a variable then go to A2;
    $i := 1 + i$; go to A1;
A2: if $a[i + 2]$ is a variable then go to A3;
    $i := 3 + i$; go to A1;
A3: if $p(a[i + 3)] \leqslant p(a[i + 1])$ then go to A4;
    $i := 2 + i$; go to A1;
A4: compile ($k$, $a[i]$, $a[i + 1]$, $a[i + 2]$);
    if a[$i - 1$] is a left-hand parenthesis $\wedge$
    $a[i + 3]$ is a right-hand parenthesis
    then
begin $a[i + 3] := Tk$;
            $t := 4$;
            $i := 1 + i$;
            $j := i - 3$ end
else begin $a[i + 2]: = Tk$;
         $t := 2$;
         $j := i - 1$ end;
```

```
for $j := j$ step $-1$ until $q$ do $a[j + t] := a[j]$;
$q := q + t$; $k := 1 + k$;
if $i < q$ then $i := q$; go to $A1$;
$A5$: end
```

References

In the reference list, the following abbreviations are used:

PK	*Problemy Kibernetiki*—irregularly appearing Soviet journal. English translation of some volumes published by Pergamon Press, New York.
SZI	*Stroje na Zpracování Informaci*—irregularly appearing Czechoslovak journal.
PZAM	Prace ZAM—series of reports of The Institute for Mathematical Machines, Warsaw, Poland.
CAP	Conference on Automatic Programming Methods, held at Warsaw, Poland, September, 1961.
FRDC	Frontier Research on Digital Computers, John W. Carr III and Mary Dale Spearman, eds., Univ. of California, Berkeley, California, 1959.
R	following the reference means that the paper has been published in Russian.
P	following the reference means that the paper has been published in Polish.

1. Arsentieva, N. G., On some transformations of programming schemes, *PK* **6**, 59–68 (1960) R.
2. Borowiec, J., Translation of arithmetic formulae, *CAP*, in *Algorytmy* **1**, 37–56 (1962).
3. Capla, V. P., A computer survey of the Soviet Union, *Datamation* **8**, No. 8, 57–59 (1962).
4. Carr, J. W., III, Bibliography on Soviet computer literature, *FRDC* **2**.
5. Carr, J. W., Report on a return visit to the Soviet Union, *FRDC* **2**.
6. Chorafas, D. N., *Programming Systems for Electronic Computers*, p. 94, Butterworths, London (1962).
7. Computing Reviews Bibliography: 3, *Comp. Rev.* **2**, 212–214 (1961).
8. Čulik, K., On languages generated by some types of algorithms, *CAP*, published in Proceedings of Munich Congress of IFIP, 1962.
9. Daugavet, O. K. and Ozerova, E. F., Programming programme of compiling type, *Zhur. Vych. Mat. i Mat. Fiz.* **1**, 747–748 (1961) R.
10. Ershov, A. P., Programming of arithmetical operators, *Doklady Akad. Nauk S.S.S.R.* **118**, 427–430 (1958), transl. in *Comm. ACM*, **1**, No. 8 (1958).
11. Ershov, A. P., *Programming programme for B.E.S.M.*, Izdatel'stvo Akademii Nauk, Moscow (1958) R, transl. by M. Nadler (1959), Pergamon Press, New York.
12. Ershov, A. P., The work of the Computing Centre of the Academy of Sciences of USSR, *Proceedings of the International Symposium on Mechanization of Thought Processes*, p. 259, H.M.S.O., London (1959)
13. Ershov, A. P., Operator algorithms, *PK* **3**, 5–48 (1960); **8**, 211–235 (1962) R.

14. Ershov, A. P., Main principles of construction of the Programming Programme in the Mathematical Institute of the Siberian Division of the Academy of Sciences USSR, *Siberian Math. Zhur.* **2**, 835–852 (1961) R.
15. Ershov, A. P., Kamynin S. S., and Lyubimskii E. Z., Automation of programming, *Trudy 3-go Vses. Matematicheskogo s'ezda* **2**, 74–76 (1956) R.
16. Ershov, A. P., Kozhukhin, G. I., and Voloshin Yu. M., *Input language for automatic programming system*, Computation Centre of the Siberian Division of the Academy of Sciences USSR (1961) R. translation: Yershov, A. P., Kozhukhin, G. I., and Voloshin, U. M. (1963); *Input Language System of Automatic Programming*, Academic Press, New York.
17. Fiałkowski, K., Swianiewicz, J., ZAM-2 Computer description and Programming in the language SAS, *PZAM* **C3**, Warsaw, (1962) P.
18. Fedoseev, V. A., Methods of automatic programming for computers, *PK* **4**, 69–94 (1960) R.
19. Glushkov, V. M., On a certain method of automation of programming, *PK* **2**, 181–184 (1959) R.
19a.Gosden, J. A. (ed.), Report of a visit to discuss common programming languages in Czechoslovakia and Poland, *Comm. ACM* **6**, No. 11, 660–662 (1963).
20. Greniewski, M., Symbolic modificators code for the EMAL-2 computer, *CAP*.
21. Greniewski, M., Algorithmic language, compiler and logical design of computers, *Proc. IFAC Symp., Moscow*, 1962.
22. Greniewski, M., Turski, W., Beschreibung der Sprache KLIPA, *Wiss. Z. Tech. Univ. Dresden*, **12**, Heft **1**, 64–68 (1963).
23. Greniewski, M., Turski, W., External language KLIPA for URAL-2 computer, *Comm. ACM* **6**, No. 6, 321–324 (1963).
24. Iliffe, J. K., The use of the Genie system in numerical calculations, *Ann. Rev. Autom. Programming* **2**, 1–28 (1961).
25. Kalmár, L., A contribution to the translation of arithmetical operators (assignment statements) into machine language of the computer M-3, *CAP*.
26. Kämmerer, W., *Ziffernrechenanlagen*, Akademie Verlag, Berlin, 1960.
27. Kamynin, S. S., Lyubimskiĭ, E. Z., and Shura-Bura, M. R., On automation of programming with the help of a programme that programmes, *PK* **1**, 135–171 (1958) R.
28. Keldysh, M. V., Lyapunov, A. A., and Shura-Bura, M. R., Mathematical problems of the theory of computers, *Vestnik Akad. Nauk S.S.S.R. No.* **11**. 16–37 (1956), R.
29. Kindler, E., Simple algorithm for the programming of arithmetic expressions, *SZI* **8**, 143–154 (1962).
30. Kindler, E., Matrix inversion on computers with fixed point operations, *SZI* **8**, 135–142 (1962).
31. Kitov, A. I., *Electronic digital computers*, Sovetskoje Radio, Moscow (1956) R. (partially translated in *FRDC* 2).
32. Kitov, A. I., and Krinitskiĭ, N. A., *Electronic Digital Computers and Programming*, Fizmatgiz, Moscow (1961) R. translation: *Electronic Computers*, Pergamon Press, New York, in press.
33. Klouček, J., and Vlček, J., Ein Entwurf des symbolischen Systems für die Formulierung der Aufgaben auf dem Gebiete der Verarbeitung von ökonomischen Daten, *SZI* **8**, 181-188 (1962) (in Czech).
34. Korobov, N. M., On some problems of equal density distributions, *Izv. Akad. Nauk S.S.S.R. Ser. Mat.*, **14**, No. 3. 215–238 (1950) R.

35. Korolyuk, V. S., On the address algorithm concept, *PK* **4**, 85–110 (1960) R.
36. Kovalev, N., quoted in *Computers and Automation* **11**, No. 9, 7 (1962).
37. Kozmidiadi, V. A., Chernyavskiĭ, V. C., On some concepts of the theory of mathematical machines, *Voprosy teorii mat. mashin* **2**, 128–143 (1962) R.
37a. Krinitskiĭ N. A., Mironov, G. A., and Frolov, G. D., *Programming*. Fizmatgiz, Moscow, 1963, R.
38. Kurochkin, V. M., A lecture to the Warsaw Conference, *CAP*.
39. Laboratoř matematických strojů, *SZI* **1**, (1953) This issue is devoted to principles of programming and computers (in Czech).
40. Lavrov, S. S., On memory economy in closed operator scheme, *Zhur. Vych. Mat. i Mat, Fiz.* **1**, 678–701 (1961) R.
41. Lyapunov, A. A., On logical schemes of programmes, *PK* **1**, 46–74, (1958) R.
42. Lyubimskiĭ, E. Z., Arithmetical block in PP-2, *PK* **1**, 178–182 (1958) R.
43. Lyubimskiĭ, E. Z., Proposed alterations in ALGOL-60, *Zhur. Vych. Mat. i Mat. Fiz.* **1**, 361–364 (1961) R.
44. Lukaszewicz, L., SAKO—An automatic coding system, *CAP*; published in *Ann. Rev. Autom. Programming* **2**, (1961).
45. Lukaszewicz, L., and Mazurkiewicz A., Automatic coding system SAKO; Part I: Description of the language, *PZAM* **C2** (1961) P.
46. Lukhovitskaya, E. S., Logical conditions block in PP-2, *PK* **1**, 172–177 (1958) R.
47. Markov, A. A., Theory of algorithms, *Trudy Mat. Inst. Akad. Nauk S.S.S.R.* **42** (1954) R.
48. Martynyuk, V. V., On the symbolic address method, *PK* **6**, 45–58 (1961) R.
49. Mazurkiewicz, A., Arithmetic subroutines and formulae in SAKO, *CAP*; published in *Ann. Rev. Autom. Programming* **2** (1961).
50. Nováková, M., and Vlček, J., Method of programming the simplex-method procedure on a digital computer, *SZI* **8**, 171–179 (1962).
51. Podlovchenko, R. I., On transformation of programming schemes and its application to programming, *PK* **7**, 161–168 (1962) R.
52. Sedlák, J., A programme for investigation of solutions of ordinary differential equations, *SZI* **7**, 99–117 (1959) R.
53. Shreĭder, Yu. A., Programming and recursive functions, *Voprosy teorii mat. mashin* **1**, 110–126 (1958) R.
54. Shreĭder, Yu. A., On concepts of generalized programming, *Voprosy teorii mat. mashin* **2**, 122–127 (1962) R.
55. Shtarkman, V. S., Block of economy of working cells in PP-2, *PK* **1**, 185–189 (1958) R.
56. Shurygin, V. A., and Yanenko, N. N., Realization of algebraic-differential algorithms on an electronic (digital) computer, *PK* **6**, 33—43 (1961) R.
57. Sobolev, S. L., A lecture to the Warsaw Conference, *CAP*.
58. Stogniĭ, A. A., Principles of a specialized programming programme, *PK* **2**, 185–189 (1959) R.
59. Stogniĭ, A. A., Solution of a problem connected with differentiation of a function with the help of a digital computer, *PK* **7**, 189–199 (1962) R.
60. Svoboda, A., Application of Korobov's sequence in mathematical machines, *SZI* **3**, p. 61 (1955) (in Czech).
61. Swianiewicz, J., XYZ Computer, Programming in Machine Language, SAB, SAS, and SO systems. *PZAM* **C1**, Warsaw (1961) P.
62. Swianiewicz, J., and Sawicki S., SAKO-translation, *CAP*.

63. Szorc, P., Subroutines in SAKO, *CAP*.
64. Trifonov, N. P., and Shura-Bura, M. R. (eds.) *Automatic Programming System* Fizmatgiz, Moscow (1961) R.
65. Turski, W., Man-Computer Communication, lecture delivered to Top Management Conference, Warsaw (1962).
66. Turski, W., Possible astronomical use of digital computers, *Postepy Astronomii* **11**, 147–161, (1963) P.
67. Vyazalov, L. H., Morozov, Yu. I., An auxiliary standard programme for checking programmes, *PK* **6**, 59–67 (1961) R.
68. Voloshin, Yu. M., *Automatic Programming Bibliography*, Siberian Division of the Academy of Sciences USSR, Novosibirsk (1961).
69. Yanenko, N. N., Reduction of a system of quasi-linear equations to a single quasi-linear equation, *Uspekhi Mat. Nauk* **10**, *Vyp.* **3**, (1955) R.
70. Yanov, Yu. I., On matrix schemes, *Doklady Akad. Nauk S.S.S.R.* **113**, 283–286 (1957) R, (translated in *FRDC* **2**).
71. Yanov, Yu. I., On the equivalence and transformations of programming schemes, *Doklady Akad. Nauk S.S.S.R.* **113**, 39–42 (1957) R. (translated in *FRDC* **2**).
72. Yanov, Yu. I. On logical schemes of algorithms, *PK* **1**, 75–127 (1958) R.