MEMORANDUM
RM-5424-PR
OCTOBER 1967

A SURVEY OF SOVIET WORK I
THE THEORY OF COMPUTER PROGRAMMIN

Robert A. DiPac

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
OCT 1967		**00-00-1967 to 00-00-1967**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Survey of Soviet Work in the Theory of Computer Programming	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Rand Corporation,Project Air Force,1776 Main Street, PO Box 2138,Santa Monica,CA,90407-2138**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**147**	

MEMORANDUM
RM-5424-PR
OCTOBER 1967

A SURVEY OF SOVIET WORK IN THE THEORY OF COMPUTER PROGRAMMING

Robert A. DiPaola

This research is supported by the United States Air Force under Project RAND—Contract No. F11620-67-C-0015—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 90406

Published by The RAND Corporation

PREFACE

The aim of this survey is to familiarize Western readers with some of the more important Soviet efforts concerning a mathematical theory of computer programming. It has grown out of the attempt to secure careful surveys of Soviet research in cybernetics as the Soviets understand this term. Accordingly it is included within the current Soviet Cybernetics Technology Project.

In selecting the mathematical theory of computer programming, featuring the theory based on the operator method of A. A. Lyapunov, we have been governed by the following considerations:

1. The material reviewed should constitute a serious, continuing Soviet effort.

2. The material should contain a sizable mathematical component.

3. The material should in some significant sense be relevant to applications.

4. The survey should not present research that is already familiar to Western experts, and hence should pursue topics for which there is no strict analogue in the West.

The survey makes use of some concepts and results of the theory of algorithms, lattice theory, and the theory of categories, such as a reader may find in Refs. [3], [5], [23], and [26]. It is to be emphasized, however, that

the requisite level of knowledge of these disciplines is quite elementary, since it is only the basic definitions and theorems of each that are used.

SUMMARY

This report is about Soviet efforts to develop a mathematical theory of computer programming. For the most part, the study traces the development of the theory which has stemmed from the operator method of programming of A. A. Lyapunov from its starting point in terms of program schemes designed to represent specific problem-solving algorithms to its algebraic formulation in terms of the theory of categories. The latter part of the report is concerned with the attempt by various Russian authors to adapt graph theory and the theory of algorithms to provide tools for the construction of superior programming languages. Illustrative examples as well as criticism of the theory are included, and the relation of the theory to automatic programming systems is discussed.

ACKNOWLEDGMENTS

We have benefited from numerous comments and suggestions, some of a technical or editorial nature, others having to do with the translation of refractory Russian sentences. Among those we wish to thank are Paul Armer, Wade Holland, Cliff Shaw, Willis Ware, Meredith Westfall, and particularly Roger Levien. Also, we wish to thank the typists, especially Mrs. Joy Taylor, for their patient execution of the painful job of typing.

CONTENTS

A SURVEY OF SOVIET WORK IN THE THEORY OF COMPUTER PROGRAMMING

1. INTRODUCTION

This study is concerned principally with a mathematical theory that has its source in one of the more extensive approaches to computer programming in the Soviet Union, the operator method of A. A. Lyapunov. The method is best regarded as an alternative to flowcharting as a means of analyzing and representing an algorithm preparatory to programming. The counterpart in this approach of a finished flowchart is a "logical scheme of a program," or a "program scheme." There is a difference in form: A flowchart is a planar arrangement of boxes with connecting arrows, while a scheme is a linear string of symbols called "operators." Thus, a scheme is linguistic rather than diagrammatic, that is, it represents an algorithm as a linear sequence of letters that symbolize various arithmetic or logical operations. These operators are indexed with various letters that generally vary over some finite set, thereby indicating the range of objects upon which the operator acts. For example, if (a_{ij}) and (b_{jk}) are n × n matrices, the symbol "A_{ijk}" could signify the operation of forming the product $a_{ij} \cdot b_{jk}$. Here each of i, j, k varies from 1 to n. Indices also appear in operators to specify which parameters occurring in some given operation are to assume in succession some finite range of values upon satisfaction of some logical condition. This corresponds to the use of

readdressing instructions in a program to govern execution of a loop. Flowcharting is the more basic and heuristic procedure; that is, in constructing a program scheme of a complex program one would almost inevitably find himself drawing diagrams, or flowcharts, as an aid. If one were, on the other hand, to develop a theory about flowcharts, he would presumably proceed by first replacing the blueprint format of a flowchart with a more amenable formalism, much like that of program schemes.

1.1. Description of the Method.

Underlying the procedure is the conception of a program as a complex operator, acting upon the contents of the storage cells of a computing machine, and representing a sequence of operators of several different types. The basis for this view is the fact that the instructions of a program resolve conveniently into several groups, within each of which the prescribed operations are rather homogeneous. Examples of such groups of operations are arithmetical, logical and control, readdressing, dispatch, and the like. Each of the groups of homogeneous operations may be labeled as an "operator," and the arrangement of the operators in some order specifies the action of the program. This provides the basis for a formal description of the programming process, and the need and the opportunity arise for making this description more complete by a thorough classification of operators, for the development

of general methods, and for the consideration and evaluation of different arrangements of the operators—all toward the end of achieving an adequate formalization of programming.

The following classification of the "standard" operators, together with their common designation, is given by Lyapunov [21] and, with minor modifications, it has generally been adopted for use in standard Soviet texts on electronic computing machines [20]:

A, arithmetical operator—prescribes calculations

P, logical operator—evaluates a logical condition

F, readdressing operator—alters the value of parameters indexing some operator; corresponds to those instructions in a program that change the addresses appearing in some instructions.

I, forming operator—generates the initial form of some operator in a program.

Φ, restoring operator—restores the initial form of some operator by restoring the initial values of parameters indexing the operator.

Z, transfer operator—transfers numerical data from one location to another. These operators are also written as "[a → b]," where a is the data to be transferred and b is the location to which a is transferred.

All other operators are described as "nonstandard" and are frequently designated by H.

Let us consider a simple example. A program is to be written to compute the function

$$\varphi(x) = f(x^2 + x - 1),$$

where

$$f(y) = \begin{cases} y^3 - 1 & \text{if } |y| \leq 1, \\ 2y - 1 & \text{if } 1 < |y| \leq 2, \\ \frac{y^5}{8} - 1 & \text{if } |y| > 2. \end{cases}$$

One proceeds directly to construct a logical scheme for the program as follows. First the operators are defined:

Π_0 – introduces the program.

A_1 – transforms the initial data into binary form

A_2 – computes the function $y = x^2 + x - 1$.

P_3 – tests the condition $1 - |y| \geq 0$. If this condition is satisfied, pass to operator A_4. If the condition tested by P_3, or, briefly, condition P_3, is not satisfied, transfer to operator P_5.

A_4 – computes $f(y) = y^3 - 1$.

P_5 – tests the condition $2 - |y| \geq 0$. If P_5 is not satisfied, then transfer to operator A_7.

A_6 – computes $f(y) = 2y - 1$.

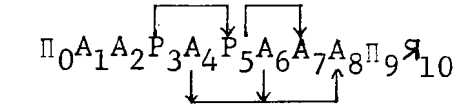
A_7 – computes $f(y) = \frac{y^5}{8} - 1$.

A_8 – converts the result of the computation into decimal form.

Π_9 – punches out the result.

$Я_{10}$ – stops the machine.

The logical scheme of the program is



The passage, of course, is from left to right.

Here the upper arrows designate the order of execution of the operators upon satisfaction of some logical condition. For instance, the arrow above "P_3" means that control is transferred to P_5 if P_3 does not hold. Otherwise, control passes to A_4. The lower arrows indicate unconditional transfer to the operator to which the arrow leads following execution of the operator from which the arrow originates.

In order to linearize the operator representation, the upper arrows are replaced by a pair of half-brackets having the same index. The left half-bracket is situated immediately to the right of the operator from which the arrow originates, and the right half-bracket is situated immediately to the left of the operator to which the arrow leads. The lower arrows are replaced by (1) the false proposition denoted by "0," to the right of the operator from which the arrow originates; (2) an immediately following indexed left half-bracket; and (3) a right half-bracket, having the same index as the left, to the immediate left of the operator to which the arrow leads. It is understood that no distinct pairs of half-brackets have the same index. The scheme thus becomes

$$\Pi_0 A_1 A_2 P_3 \underset{1}{\llcorner} A_4 0 \underset{21}{\llcorner\lrcorner} P_5 \underset{3}{\llcorner} A_6 0 \underset{43}{\llcorner\lrcorner} A_7 \underset{2}{\lrcorner} \underset{4}{\lrcorner} A_8 \Pi_9 Я_{10} .$$

The operators of this scheme can be programmed separately and the resulting subprograms fitted together in accordance with the plan of the scheme. At this stage the program is ready to be punched and run on the machine.

1.2. Transformations of Program Schemes

The theory of operator programming is, more than anything else, an investigation of the effects of various types of transformations on program schemes. A major aim is to classify those transformations that give rise to a scheme equivalent to some given scheme in the sense of prescribing a program that will solve the same problem, or in some stronger sense, but that exhibits some technical advantage. For example, transformations on the logical conditions present in a logical scheme according to various rules of the propositional calculus may lead to a simplification in the form of these conditions that facilitates the actual programming. Again, transformations on the parameters indexing the arithmetical operators may be designed to take advantage of some feature of the problem, such as symmetry in the case of matrix calculations, and lead to a new indexing that makes more economical use of the memory of the machine.

1.3. Programming Programs

The first Soviet attempts at automatic programming, resulting in "programming programs" (PP), were based on the operator method. Indeed, Turski in [33] asserts: "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 feasibility of a PP lies in this: the basic structure of the operators

of each of the types A, P, Z, F, and Φ does not vary with the algorithm to be programmed. This basic structure may be programmed as a collection of "generalized instructions" that are translated into machine language. Thus, a program is constructed which, when supplied with the encoded form of a logical scheme for a program and with the data realizing the operators in the given scheme, will assemble a program prescribed by the logical scheme. For example, a generalized instruction associated with a logical operator could be: "Evaluate a given logical function of several arguments and, depending on whether it is equal to zero or unity, execute one or another generalized instruction of the algorithm."

The information about the operators which, in coded form, constitutes part of the input to a PP may be summarized as follows.

1. A–operator: an ordered list of the calculation formulas to be used.

2. P–operator: the logical condition to which P refers, specifying its constituent elementary logical conditions and the indices of the operators that are to be executed depending upon the truth or falsity of the condition.

3. F–operator: the operators whose addresses are to be modified by F and the address of the corresponding parameter.

4. Z—operator: the quantities to be transferred to the standard cells and the addresses of the operators in which these quantities are to be replaced by the contents of the standard cells.

5. Ф—operator: the addresses of the operators to be restored.

6. H—operator: the list of machine instructions comprising H. Thus nonstandard operators receive a different treatment from that of the other types of operators. The generalized instruction corresponding to H is typically of the form: "Carry out a given list of elementary machine instructions." These machine instructions are written manually by the programmer.

Additional input data for the PP relates to the assembled program as a whole:

1. A table of the contents of the elementary logical conditions.

2. A table describing the indexing of the initial data of the problem and the transformations that may lead to a new indexing of this information.

3. A table of auxiliary constants.

4. A table describing the working cells.

5. A table of storage distribution indicating the addresses in which quantities used in calculations are stored.

6. The address of the cells at which the program begins.

7. The permissible length of the program.

8. The address of the cell to which control should be transferred at the end of the program.

The reader will observe that the above information includes implicitly the logical scheme of the program.

All of this information is encoded in numerical form on program sheets, from which it is transferred to punched cards that are read into the machine.

The reader can find extensive descriptions of programming programs based on the operator method in [6] and [19]. Some critical comments on PPs are included in Part 4.

1.4. Organization of the Survey

Part 2 traces the development of the theory of operator programming. First, a presentation is given of the initiating paper of Lyapunov [21]. The subsequent section is concerned with the study of transformations of program schemes and divides naturally along logical and nonlogical lines. The work of Yanov on the transformation of the logical conditions present in a program scheme is covered, while the discussion of nonlogical transformations is based chiefly on papers of Podlovchenko and Arsenteva [2, 27, 28].

A paper of Podlovchenko [29] is presented as a recapitulation of basic terminology and notions of the theory from a more abstract point of view. It may serve as a summary of the notions so far developed.

The algebraic treatment of programming is described in 2.4. In this treatment the initiating paper is again due to Lyapunov [22]; two papers by Glebov [14], [15] are also included. Again the motive is the characterization of transformations operating on program schemes, but now the treatment is quite abstract; the framework is that of lattice theory and the theory of categories.

Part 3 of the survey is concerned not with the theory having the operator methods as its source, but with the attempt, initiated by Kaluzhnin in [18], to develop methods and a formalism aptly suited for the description of algorithms to be programmed, and with the related and derived efforts of Ershov [10, 11] to reformulate the theory of algorithms to render it more suitable for the study of problems arising in the theory of programming. In this connection, the work of Zaslavskii [36] is also mentioned.

Our aim has not been to cover, or even mention, all Soviet papers that touch on the pertinent topics, but rather to select representative papers and organize them into a coherent framework. A number of the papers are treated in some detail so that the reader can become fully acquainted with the mathematical setting and methods should he wish to conduct similar research or pursue open problems. On the other hand, the technical developments are always preceded by a informal statement of the aims, main results, and methods of the papers

sufficiently complete to allow the reader to omit the details should he be so disposed. Occasional critical comments are interspersed throughout the study. In the final part, Part 4, summarizing comments are given, and several questions are posed concerning the further development and use of the theory.

2. THE OPERATOR THEORY OF PROGRAMMING

2.1. Informal Statement of the Basic Notions

The aim of this section is to introduce a formalism that provides the means for a symbolic description of programs composed to solve particular problems and to indicate the possible lines that a theoretical development might take. The main source of the exposition is due to A. A. Lyapunov [21].

The initial representation of an algorithm in this formalism is a calculation scheme—a finite sequence of symbols, each one of which is either an operator A_i or a logical condition p_j. One should think of the operators as acting on the contents of some cells in the memory of a computer. A sequence of juxtaposed operators is termed their product. The action of an operator frequently depends on the parameters with which it is indexed. To illustrate, we construct a calculation scheme for the product $C = A \cdot B$ of two n x n matrices A and B. Let operator A_{ijk} represent the multiplication $a_{ij}b_{jk}$ of elements a_{ij} and b_{jk} and addition of the result to the contents of the cell in which c_{ik} is to be stored. Then

$$\prod_{j=1}^{n} A_{ijk}$$

is the scheme of calculation for element c_{ik}, and

$$\prod_{k=1}^{n} \prod_{j=1}^{n} A_{ijk}$$

is the scheme of calculation for the i-th row of C = A · B. Finally, $\prod_{i=1}^{n} \prod_{k=1}^{n} \prod_{j=1}^{n} A_{ijk}$ is the calculation scheme for forming the product A · B.

A calculation scheme is obviously a primitive, skeletal representation of an algorithm. It makes no provision for loops—the iterated execution of some sequence of operators in some fixed order. A program scheme, as defined below, is a development of a calculation scheme in which the role of the control operators is more fully specified.

The control operators are broken down into the following categories: (1) readdressing operators, (2) restoration operators, (3) formation operators, (4) parameter change operators, (5) transfer operators, (6) logical condition switching operators, (7) parameter introducing operators. (Such a list is, of course, in part conventional, and a little work could result in one having 17 categories instead of 7.)

A program scheme, then, is a finite sequence of symbols, each of which is either an operator, denoted by indexed capital Roman letters, or a logical condition $p_i\uparrow^j$, or $p_i\downarrow^j$, consisting of a proposition p_i and an indexed upward arrow $\uparrow^j$, or an indexed downward arrow $\downarrow^j$. Frequently, the logical conditions present in a scheme take the form of such statements as "p(i>j)", which is true if i>j, and false otherwise.

The manner of execution of a program scheme, for some assignment of values to the parameters present in the scheme, is as follows. Begin at the extreme left and proceed to the right until an operator A_i or a logical condition $p_j\overset{k}{\uparrow}$ is encountered. If the encountered symbol is an operator A_i, it is executed and the movement to the right is resumed; if it is a logical condition and $p_j = 1$ (= truth) for the given assignment, again move to the right, if the value of p_j is 0, and then "branch out" and enter the scheme immediately to the right of $\overset{k}{\downarrow}$. This procedure may go on indefinitely or may involve execution of the operator STOP, at which point the process comes to a halt. (One can be more precise about the syntax of program schemes, but at this point, such rigor seems misplaced.)

The program scheme based on the above calculation scheme for the multiplication of two matrices is as follows:

$$\overset{1,2,3}{\downarrow} A_{ijk}F(j)p(j>n)\overset{1}{\uparrow}F^{-n}(j)F(k)p(k>n)\overset{2}{\uparrow}F^{-n}(k)F(i)$$

$$p(i>n)\overset{3}{\uparrow} \text{STOP.} \quad \text{Starting values:} \quad i = j = k = 1.$$

Here, an operator of the form $F(m \cdot i)$ means "increment parameter i by m units." $F(m \cdot i)$ is called a readdressing or restoration operator, according as $m > 0$ or $m < 0$. We write $F^m(k)$ instead of $F(mk)$.

In some cases, more than one algorithm may solve a given problem. These give rise to distinct calculation

schemes, and these, in turn, to distinct program schemes. Comparison of these schemes could hopefully lead to selection of the better program. "Better" here, of course, has a contextual, but obvious, sense. The scheme, or resulting program, may be better than another in facilitating programming or allowing for superior machine operation.

As an example, let us consider the problem of calculating the determinant of an n x n matrix (a_{ij}). A is assumed to be in upper triangular form, and the determinant is computed as the product of the diagonal elements. So, for $i > j$ we have the operators

B_{ij}: evaluate $c = a_{ji}/a_{ii}$ and store.

A_{ijk}: evaluate $a_{jk}' = a_{jk} - ca_{ik}$ and store in the location of a_{jk}.

D_m: multiply a_{11} by a_{mm} and store in the location of a_{11}. Calculation schemes:

$$(1)\quad \left(\prod_{i=1}^{n} \prod_{j=i+1}^{n} B_{ij} \prod_{k=1}^{n} A_{ijk} \right) \prod_{m=2}^{n} D_m,$$

$$(2)\quad \left(\prod_{i=1}^{n} \prod_{j=1}^{n} B_{ij} \prod_{k=1}^{n} p(i>j)A_{ijk} \right) \prod_{m=2}^{n} D_m.$$

Scheme (1) gives rise to the following program scheme:

$$\overset{2,3}{\downarrow} B_{ij} \overset{1}{\downarrow} A_{ijk} F(k)p(k>n) \overset{1}{\uparrow} F^{-(n-i)}(k)F(j)p(j>n) \overset{2}{\uparrow}$$

$$F^{-(n-i+1)}(j)F(i)p(i>n) \overset{3}{\uparrow} \overset{4}{\downarrow} D_m F(m)p(m>n) \overset{4}{\uparrow} \text{STOP}$$

Scheme (2) gives rise to

$$\overset{3,4}{\downarrow} B_{ij} \overset{2}{\downarrow} p(i<j) \overset{1}{\uparrow} A_{ijk} \overset{1}{\downarrow} F(k)p(k>n) \overset{2}{\uparrow} F^{-n}(k)F(j)p(j>n) \overset{3}{\uparrow}$$

$$F^{-m}(j)F(i)p(i>n) \overset{4}{\uparrow} \overset{5}{\downarrow} D_m F(m)p(m>n) \overset{5}{\uparrow} \text{STOP.}$$

The first of these schemes contains more orders than the second, since F(i) must operate on all instructions of the form $F^{-(n-i)}(k)$ and $F^{-(n-i+1)}(j)$. On the other hand, the second requires more cycles since F(k) and F(j) readdress without any computation taking place until $k = n + 1$ or $j = i + 1$. A choice is to be made between these two schemes on the basis of some criterion.

The tone and style of Lyapunov's paper [21] are loose and heuristic. There is no substantive mathematics in the paper, but neither was there meant to be. Questions are raised that form the basis for papers that follow. For example, it is noted that the question of equivalent rearrangements of program schemes should be investigated, one arrangement possibly leading to the construction of a "better" program than another. This leads directly to the work of Yanov, which is discussed in the next section. Also,

Lyapunov considers a number of other examples such as a detailed example of solving the Dirichlet problem using a varying net mesh. The paper also describes a technical device (logical scales) used to change the logical conditions in a program scheme in the course of the computation.

2.2. Transformations on Program Schemes

2.2.1. Transformations of the Logical Conditions.

In order to treat adequately and precisely the question of rearrangement of program schemes by changes in their logical conditions, the formal structure of these schemes must be more adequately defined. This is achieved in a paper by Yu. I. Yanov [35]. This paper introduces a formal system, an interpretation of which is the program schemes of Lyapunov. In this connection, the questions investigated are of a metalogical character, questions of the equivalence of schemes under a certain system of logical transformations and the completeness of this system. The letters that represent operations, the operator symbols, now assume the status of primitives, and the logical scheme looks much like a program scheme except that the up and down arrows have been traded in for left and right half brackets, respectively.

Generally speaking, the problems treated by Yanov amount to considering the effect on a logical scheme of transformations of the logical variables contained therein. Two such problems are:

(1) Under a suitable definition of equivalence, can the equivalence of two logical schemes be decided by some uniform effective procedure?

(2) Can a system of logical transformations be devised that will generate all logical schemes equivalent to a given one?

Yanov solves these problems in the affirmative for two kinds of logical schemes, a one-dimensional or linear type and a two-dimensional or matricial (rectangular) variety. He also considers the question of the equivalence of a linear to a matricial scheme, and shows how to pass from one to another.

We assume that we have the usual apparatus of the propositional calculus. If α is a propositional variable and $\lfloor_i$ is a left half-bracket, then $\alpha\lfloor_i$ is a <u>logical condition</u>. A logical scheme of an algorithm is a finite sequence of symbols composed of operator symbols $A_1, A_2, \ldots$, logical conditions $\alpha\lfloor_i, \beta\lfloor_j, \ldots$, and right half-brackets $\rfloor_i, \rfloor_j, \ldots$, such that for each left half-bracket $\lfloor_i$ with index i appearing in the sequence there is exactly one right-half bracket $\rfloor_i$ with index i, and conversely. It is understood that an operator has at most one occurrence in a logical scheme.

We must define what makes a (linear) logical scheme $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$ "go", that is, the procedure by which the scheme is executed. Let $\Delta_1, \Delta_2, \ldots, \Delta_{2^k}$ denote the 2^k possible assignments of values to

the logical variables $p_1, p_2, \ldots, p_k$, and suppose an infinite sequence $\Delta_{s_1}, \Delta_{s_2}, \ldots$, of such assignments is given.

Step 1. Assign Δ_{s_1} to $p_1, p_2, \ldots, p_k$ and mark the left-most symbol in $\mathcal{A}$.

Suppose that ℓ steps have been carried out with the result that the operators $A_{i_1}, A_{i_2}, \ldots, A_{i_{m-1}}$ have been marked and that some symbol S of $\mathcal{A}$ has been marked in the ℓ-th step. Step $(\ell + 1)$ is defined as follows:

(a) If S is an operator A_{i_m}, adjoin it to $A_{i_1}, A_{i_2}, \ldots, A_{i_{m-1}}$ on the right, giving $A_{i_1}, A_{i_2}, \ldots, A_{i_m}$; assign $\Delta_{s_{m+1}}$ to $p_1, p_2, \ldots, p_k$; and mark the symbol in $\mathcal{A}$ to the immediate right of A_{i_m}.

(b) If $S = \alpha(p_1, \ldots, p_k)\lfloor_i$, then if $\alpha(\Delta_{s_m}) = 1$, the symbol to the immediate right of $\alpha(p_1, \ldots, p_k)\lfloor_i$ is marked, whereas if $\alpha(\Delta_{s_m}) = 0$, the symbol in $\mathcal{A}$ to the immediate right of $\rfloor_i$ is marked.

(c) If $S = \rfloor_i$ for some i, the symbol to the immediate right of $\rfloor_i$ in $\mathcal{A}$ is marked.

No symbol is marked in step $(\ell + 1)$ except as specified by (a), (b) or (c). Otherwise, the process terminates.

Knowing the input and manner of operation of a scheme $\mathcal{A}(p_1, \ldots, p_k, A_1, \ldots, A_n)$, what is one to understand as the output or value of the scheme? Yanov chooses to define the output of $\mathcal{A}(p_1, \ldots, p_k, A_1, \ldots, A_n)$ to mean the entire sequence of calculations, including the order in which they are performed. Thus, the <u>value</u> of the scheme for a given

assignment is the sequence of marked operator symbols. The value may be infinite or it may be finite even though the scheme is executed indefinitely. In the latter case, the value is termed an empty period, and a pair of parentheses, ( ), is written to the right of the value.

As an application, consider the normal algorithm $\mathcal{A}$ over a finite alphabet A with the scheme

$$\begin{aligned} P_1 &\rightarrow Q_1 \\ P_2 &\rightarrow Q_2 \\ &\vdots \\ P_s &\rightarrow .Q_s \\ &\vdots \\ P_n &\rightarrow Q_n \end{aligned}$$

where $p_s \rightarrow \cdot\, Q_s$ is the final formula [23].

Let p_i stand for the assertion: "The word P_i goes over into the transformed word"; and let A_i represent the substitution of the word Q_i for the first appearance of the word P_i in the transformed word. With this interpretation of the logical variables p_i and the operators A_j, the scheme

$$\mathcal{A}(p_1, \ldots, p_n, A_1, \ldots, A_n) =$$

$$\underset{n+1}{\lrcorner}\,\underset{n}{\lrcorner}\;\underset{n+s-1}{\lrcorner}\;\underset{n+s+1}{\lrcorner}\;\cdots\underset{2n}{\lrcorner}\;p_1\;\underset{1}{\llcorner}\;A_1 0\;\underset{n+1}{\llcorner}\,\underset{1}{\lrcorner}\;p_2\;\underset{2}{\llcorner}\;A_2\;0\;\underset{n+2}{\llcorner}\;\cdots$$

$$\cdots\underset{s-1}{\lrcorner}\;p_s\;\underset{s}{\llcorner}\;A_s\;0\;\underset{n+s}{\llcorner}\;\cdots\;\underset{n-1}{\lrcorner}\;p_n\underset{n}{\llcorner}\;A_n\;0\;\underset{2n}{\llcorner}\,\underset{n}{\lrcorner}\;\underset{n+s}{\lrcorner}$$

represents the given normal algorithm.

Let us write $V(\mathcal{A})$ for the value of scheme a for some given assignment. Consider the scheme $\mathcal{A}(p_1, p_2, A_1, A_2) = \underset{2}{\lrcorner}p_1\underset{1}{\llcorner}A_1\underset{1}{\lrcorner}$

$P_2 \llcorner_2 A_2$. For the assignment (1,0), (0, 0), (0, 0), ..., $V(\mathcal{A}) = A_1(\ )$, whereas for the assignment (1,0), (1, 0), (1, 1), ..., $V(\mathcal{A}) = A_1A_1A_2$. Now, according to the above definition of execution, the values of the logical variables of a scheme may change only as an operator is executed. The variation of the logical variables controls the order in which the operation of the scheme is carried on. In applications, this amount of "transferring" can depend on the nature of the operators. On the theoretical level this is reflected in Yanov's notion of a shift distribution, which is central to his entire development. The manner in which the logical variables are allowed to change values should be closely connected with the operators; it proceeds as follows. A shift distribution can be thought of as a mapping of the set of operators A_i appearing in

$$\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$$

into the set of subsets

B_i of $B = \{p_1, p_2, \ldots, p_k\}$, and is denoted by $A_i - B_i$.

When, in accordance with the definition of operation of a scheme, the operator A_i is to be executed, only the variables in B_i are allowed to assume new values, although they may also maintain the same values.

More precisely, suppose that for a given assignment $\Delta_{s_1}, \Delta_{s_2}, \Delta_{s_3}, \ldots, \Delta_{s_m}, \ldots$ the scheme $\mathcal{A}(p_1, \ldots, p_k, A_1, \ldots, A_n)$ with shift distribution $A_i - B_i$ has value $A_{i_1}, A_{i_2},$

..., A_{i_m}, ... The given assignment is admissible with respect to this distribution if for all m, $\Delta_{s_{m+1}} = \Delta_{s_m}$ or $\Delta_{s_{m+1}}$ differs from Δ_{s_m} in the values of variables in B_{i_m}. For example, if $\mathcal{A}(p_1, p_2, A_1, A_2) = P_1 \lfloor A_1 \rfloor_2 \rfloor_1 A_2\ p_2 \lfloor_2$ and the shift distribution $A_1 - \{p_1\}$, $A_2 - {}^{-1}\{p_2\}$ is given, the assignments

(1) (1,0), (0,0), (0,1), (0,1), ...

and

(2) (1,0), (1,0), (1,0), (1,0), ...

are admissible, whereas the assignment

(3) (1,0), (1,0), (0,0), (0,0), ...

is not admissible.

Now, two algorithms may be said to be equivalent in several different ways. Most generally, two might be considered equivalent if they give the same (or isomorphic) outputs for the same (or isomorphic) inputs no matter how differently the process of computation is carried out; their "programs" could be very different. With this definition the general problem of deciding whether two algorithms are equivalent is, as is well known, recursively unsolvable. Since we have already stated that Yanov shows that the equivalence problem is solvable, it is to be expected that he employs a rather strong definition of equivalent schemes. Indeed, two schemes $\mathcal{A}(p_1, \ldots, p_k, A_1, \ldots, A_n)$ and $\mathcal{B}(p_1, \ldots, p_k, A_1, \ldots, A_n)$ are equivalent in Yanov's sense with respect to a shift distribution B if they have the same values for all

admissible assignments. Thus, if two schemes are equivalent for a shift distribution, they produce the same sequence of operations for all admissible inputs and all interpretations of the operator symbols.

Equivalent schemes need by no means be identical. They may, in fact, not be equivalent with respect to some other shift distribution.

The schemes

$$\mathcal{A}(p_1, p_2, A_1, A_2) = p_1\ \overline{p}_2 \underset{1}{\llcorner} \underset{2}{\lrcorner} A_1 p_2 \underset{2}{\llcorner} 0 \underset{3}{\llcorner} \underset{1}{\lrcorner} A_2 \underset{3}{\lrcorner}$$

and

$$\mathcal{B}(p_1, p_2, A_1, A_2) = \overline{p}_1 \cdot p_2 \underset{1}{\llcorner} A_2 0 \underset{2}{\llcorner} \underset{1}{\lrcorner} \underset{3}{\llcorner} A_1 p_2 \underset{3}{\llcorner} \underset{2}{\lrcorner},$$

where 0 is the false proposition, are clearly equivalent with respect to the universal distribution, that is, where $B_1 = \{p_1, p_2\} = B_2$. Two schemes that are equivalent with respect to the universal distribution are equivalent with respect to any shift distribution, since for the universal distribution all sequences of assignments are admissible.

On the other hand, the schemes

$$\mathcal{A}(p_1, p_2, A_1, A_2) = p_1 \underset{1}{\llcorner} p_2 \underset{2\ 3}{\llcorner} \lrcorner A_1 p_2 \vee (p_1 . p_2) \underset{4}{\llcorner} A_2 0 \underset{3\ 4\ 2\ 1}{\llcorner} \lrcorner\lrcorner\lrcorner$$

and

$$\mathcal{B}(p_1, p_2, A_1, A_2) = p_1 \underset{1}{\llcorner} p_2 \underset{2\ 3}{\llcorner} \lrcorner A_1 p_1 \vee (p_1 \cdot p_2) \overline{\underset{4}{\llcorner} A_2\ 0} \underset{3\ 4\ 2\ 1}{\llcorner} \lrcorner\lrcorner\lrcorner$$

are obviously equivalent with respect to the empty distribution in which only the constant sequences are admissible, but they are not equivalent with respect to the shift

distribution $A_1 - \{p_2\}$, $A_2 - \phi$. Consider the sequence

(1,1), (1,1), (1,1), (1,0), (1,0), (1,0), ...

In this case, $V(\mathcal{A}) = A_1A_2A_1$ and $V(\mathcal{B}) = A_1A_2A_1A_2A_1A_2\ A_1A_2\ldots$ Here the sequence is admissible for both $\mathcal{A}$ and $\mathcal{B}$, but the values of $\mathcal{A}$ and $\mathcal{B}$ are different. Hence, $\mathcal{A}$ is not equivalent to $\mathcal{B}$ for this distribution.

It is obvious that equivalence with respect to the empty distribution is solvable for schemes $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$ and $\mathcal{B}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$, since one needs only to examine the 2^k possible assignments. This simple fact is important in that a recurrent theme throughout Yanov's paper is this: A problem concerning two schemes $\mathcal{A}$ and $\mathcal{B}$ with respect to an arbitrary shift distribution is translated into a closely related problem concerning closely related schemes $\mathcal{A}'$ and $\mathcal{B}'$ with respect to the empty distribution. In particular, in regard to the problem of deciding the equivalence of a pair of (linear) schemes, it is demonstrated that each scheme $\mathcal{A}$ with a given shift distribution can be transformed into a certain canonical form $\mathcal{A}^{(\mu)}$, called the stationary merger of $\mathcal{A}$. The stationary merger is a function of the scheme $\mathcal{A}$ and the shift distribution; different shift distributions in general give rise to different stationary mergers. It is proved that schemes $\mathcal{A}$ and $\mathcal{B}$ are equivalent with respect to a given shift distribution if and only if their stationary mergers are equivalent with respect to the empty distribution. The basic idea of the proof is as follows: Consider a scheme $\mathcal{A}$ with a shift distribution B and the value V of $\mathcal{A}$ for some admissible sequence. By examination of all the various

possibilities allowed by the set B_i associated with operator A_i and the introduction of auxiliary variables, $\mathcal{A}$ is transformed into a scheme $\mathcal{A}^{(\mu)}$ such that if A_i is satisfied, that is, if A_i is executed for some given sequence of assignments, then A_i will be satisfied in $\mathcal{A}^{(\mu)}$ for some stationary sequence. By definition of $\mathcal{A}^{(\mu)}$ one can find an admissible sequence of assignments for which A_i is satisfied in $\mathcal{A}$; that is, all possible variation in the logical variables that would enable A_i to be satisfied in $\mathcal{A}$ have been built into the definition of $\mathcal{A}^{(\mu)}$ so that we can find a sequence of assignments such that A_i is satisfied in $\mathcal{A}$ for this sequence. This holds for each operator A_i, and hence for each value V of $\mathcal{A}$.

For our purposes, it suffices to present the definition of a stationary merger $\mathcal{A}^{(\mu)}$ of scheme $\mathcal{A}$ for a given distribution B.

Let E be an elementary expression, that is, a logical condition or an operator, in scheme $\mathcal{A}(p_1, \ldots, p_k, A_1, \ldots, A_n)$.

Let

$$E^{\otimes}_{(\mathcal{A})}(\Delta_s) = \begin{cases} 1 \text{ if E is executed in } \mathcal{A} \text{ for the sequence } \Delta_s, \Delta_s, \Delta_s, \ldots, \\ 0 \text{ otherwise,} \end{cases}$$

and $\alpha_i^1 = \max_{B_i} A_{i(\mathcal{A})}^{\otimes}(p_1, p_2, \ldots, p_k)$, $i = 1, 2, \ldots, n$.

Then

$$\mathcal{A}^{(1)} \equiv q_1 \vee \bar{\alpha}_1^1 \underset{j_1}{\lfloor} \ldots q_n \vee \bar{\alpha}_n^1 \underset{j_n}{\lfloor} \mathcal{A}(p_1, \ldots, p_k, A_1 \underset{j_1}{\rfloor}, \ldots, A_n \underset{j_n}{\rfloor}).$$

Suppose the functions $\alpha_i^\nu, i = 1, 2, \ldots, n$, and the ν–th merger $\mathcal{A}^{(\nu)}$ of $\mathcal{A}$ have been defined.

Then $\alpha_i^{\nu+1} \quad \max_{\substack{B_i, q_1, q_2, \ldots, q_n \\ i=1, 2, \ldots, n.}} A_i^{\otimes}{}_{(\mathcal{A}^{(\nu)})}(p_1, \ldots, p_k, q_1, \ldots, q_n)$,

$$\mathcal{A}^{(\nu+1)} \equiv q_1 \vee \bar{\alpha}_1^{\nu+1} \underset{j_1}{\llcorner} \ldots q_n \vee \bar{\alpha}_n^{\nu+1} \underset{j_n}{\llcorner} \mathcal{A}(p_1, \ldots, p_k, A_1 \underset{j_1}{\lrcorner}, \ldots, A_n \underset{j_n}{\lrcorner}).$$

It is clear that for $i = 1, 2, \ldots, n$ and all ν, $\alpha_i^\nu \to \alpha_i^{\nu+1}$ so a natural number μ is arrived at such that $\alpha_i^{\mu+1} \equiv \alpha_i^\mu$ for $i = 1, 2, \ldots, n$. The stationary merger $\mathcal{A}^{(\mu)}$ of $\mathcal{A}$ is defined as follows:

$$A_i' = \max_{q_1, \ldots, q_n} A_i^{\otimes}{}_{(\mathcal{A}^{(\mu)})}(p_1, \ldots, p_k, q_1, \ldots, q_n)$$

$$A_i'' = \alpha_i^\mu, \quad \text{so } A_i'' = \max_{B_i} A_i',$$

and

$$\mathcal{A}^{(\mu)} \equiv q_1 \vee \bar{A}_1'' \underset{j_1}{\llcorner} \ldots q_n \vee \bar{A}_n'' \underset{j_n}{\llcorner} \mathcal{A}(p_1, \ldots, p_k, A_1 \underset{j_1}{\lrcorner} \ldots, A_n \underset{j_n}{\lrcorner}).$$

As mentioned before, to say schemes $\mathcal{A}$ and $\mathcal{B}$ are equivalent for a shift distribution B is to say that they produce the same sequence of operations for all admissible inputs and all interpretations of the operator symbols. The area for

possible variation in equivalent schemes, then, is in the form of the logical variables. The question arises of having an effective method for generating all schemes $\mathcal{B}$ equivalent to given scheme $\mathcal{A}$ relative to some distribution B. Accordingly, Yanov supplies a list of eleven transformations such that if $\mathcal{A}$ is equivalent to $\mathcal{B}$ for some distribution, then $\mathcal{A} = \mathcal{B}$ can be derived by use of the transformation calculus. In other words, the list of transformations is complete with respect to the property of equivalence of schemes. The place of the shift distribution in the situation is taken into account by the concept of subordination of an elementary expression E to a logical function α ($E < \alpha$). First, given scheme $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, \ldots, A_n)$ with shift distribution B, define $E' = \max_{q_1,q_2,\ldots,q_n} E^{\otimes}(\mathcal{A}^{(\mu)})$, where $\mathcal{A}^{(\mu)}$ is the stationary merger of $\mathcal{A}$ relative to B. E is said to be <u>subordinate</u> to $\alpha(p_1, p_2, \ldots, p_k)$ if $E' \rightarrow \alpha$ is a tautology. Thus, if α fails to hold for a given sequence of arguments, the expression E is not executed for this sequence of assignments.

In presenting the list of transformations some stipulations are in order. The letter "A" stands for an arbitrary operator, and "1" and "0" denote the true and false proposition, respectively. In expressions $C(\mathcal{A})$ in which $\mathcal{A}$ occurs, if $\mathcal{A} = \mathcal{L}$, the operation of substitution of $\mathcal{B}$ for $\mathcal{A}$ in $C(\mathcal{A})$ giving $C(\mathcal{B})$ is defined, provided no confusion results; that is, provided there results no duplication of indices of right (left) brackets and no duplication of operators. Under

interpretation, this includes the fact that if $\mathcal{A} = \mathcal{B}$ and $C(\mathcal{A})$ is a scheme, then $C(\mathcal{B})$ is a scheme and $C(\mathcal{A}) = C(\mathcal{B})$. The list is:

I.1. $0 \lfloor_i A \rfloor_i = 0 \lfloor_i \rfloor_i$.

2. $1 \lfloor_i \mathcal{A} \rfloor_i = \mathcal{A}$.

3. $\rfloor_i \mathcal{A}\, 1 \lfloor_i = \mathcal{A}$.

II.1. $\alpha\beta \lfloor_i \mathcal{A} \rfloor_i = \lfloor_i \beta \lfloor_j \mathcal{A} \rfloor_i \rfloor_j$.

2. $\rfloor_i \mathcal{A}\, \alpha\beta \lfloor_i = \rfloor_i \rfloor_j \mathcal{A} \lfloor_i \beta \lfloor_j$.

3. $\alpha \vee \beta \lfloor_i = \overline{\alpha} \lfloor_j \beta \lfloor_i \rfloor_j$.

III. $\mathcal{A}\mathcal{B} = 0 \lfloor_i \rfloor_j \beta\, 0 \lfloor_k \rfloor_i \mathcal{A}\, 0 \lfloor_j \rfloor_k$.

IV. $\rfloor_i \rfloor_j = \rfloor_j \rfloor_i$.

V. $\alpha \lfloor_i \rfloor_i = \Lambda$, where Λ is the null expression.

VI.1. $\alpha \lfloor_i \mathcal{A} \rfloor_i \alpha \lfloor_j \mathcal{B} \rfloor_j = \alpha \lfloor_i \mathcal{A}\, \alpha \lfloor_j \mathcal{B} \rfloor_j \rfloor_i$.

2. $\alpha \lfloor_i \mathcal{A} \rfloor_j \mathcal{B} \rfloor_i \alpha \lfloor_j = \alpha \lfloor_i \mathcal{A} \rfloor_j \rfloor_i \mathcal{B}\, \alpha \lfloor_j$.

3. $\rfloor_j \mathcal{A}\, \alpha \lfloor_i \mathcal{B} \rfloor_i \alpha \lfloor_j = \rfloor_j \rfloor_i \mathcal{A}\, \alpha \lfloor_i \mathcal{B}\, \alpha \lfloor_j$.

4. $\rfloor_i \alpha \lfloor_j \mathcal{A}\, \alpha \lfloor_i \mathcal{B} \rfloor_j = \alpha \lfloor_j \mathcal{A}\, \alpha \lfloor_i \mathcal{B} \rfloor_j \rfloor_i$.

5. $\rfloor_i \alpha \lfloor_j \mathcal{A} \rfloor_j \mathcal{B}\, \alpha \lfloor_i = \alpha \lfloor_j \mathcal{A} \rfloor_j \rfloor_i \mathcal{B}\, \alpha \lfloor_i$.

6. $\rfloor_j \mathcal{A} \rfloor_i \alpha \lfloor_j \mathcal{B}\, \alpha \lfloor_i = \rfloor_j \rfloor_i \mathcal{A}\, \alpha \lfloor_j \mathcal{B}\, \alpha \lfloor_i$.

VII. $\rfloor_i \alpha \lfloor_i \mathcal{A} \rfloor_j \alpha \lfloor_j = \alpha \lfloor_i \mathcal{A} \rfloor_i \rfloor_j \alpha \lfloor_j$.

VIII. $\dfrac{\vdash \alpha \equiv \beta}{\vdash \mathcal{A}(\alpha) \equiv \mathcal{A}(\beta)}$.

IX. $\dfrac{\vdash \mathcal{A} = \mathcal{B},\ \vdash C(\mathcal{A}) = D}{\vdash C(\mathcal{B}) = D}$.

X. The pair $\llcorner_i$, $\lrcorner_i$ of half-brackets occurring in an expression may be replaced by the pair $\llcorner_j$, $\lrcorner_j$, provided no interference of half-brackets results.

XI. If $\alpha \llcorner_i$ is subordinated to β for a given shift distribution, then $\alpha \llcorner_i$ may be replaced by $\alpha\beta \llcorner_i$.

The proof of the completeness of the system of transformations proceeds by transforming a linear scheme into a canonical form of the same. Equivalent schemes are transformed into the same canonical form. We confine ourselves to presenting the definition of the canonical form of a scheme $\mathcal{A}$ relative to a given distribution.

Given $\mathcal{A}$, we put $\hat{\mathcal{A}}_0 \equiv \mathcal{A}$,

$$\hat{\mathcal{A}}_i \equiv \overline{A}_i'' \llcorner_s A_0\, \mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_i \lrcorner_s, \ldots, A_n), \quad i = 1, 2, \ldots, n,$$

where A_0 is an operator distinct from each of $A_1, A_2, \ldots, A_n$.

For $i = 0, 1, \ldots, n$ and $j = 1, 2, \ldots, n$ we define

$$\alpha_{ij}(\Delta_s) = \begin{cases} 1 \text{ if } A_j \text{ is the first operator of the value of } s \\ \hat{\mathcal{A}}_i \text{ for sequence } \Delta_s, \Delta_s, \ldots, \Delta_s, \ldots, \\ 0 \text{ otherwise;} \end{cases}$$

$$\alpha_{i,n+1}(\Delta_s) = \begin{cases} 1 \text{ if the value of } \hat{\mathcal{A}}_i \text{ for the sequence } \Delta_s, \Delta_s, \\ \ldots, \Delta_s \text{ is an empty period,} \\ 0 \text{ otherwise;} \end{cases}$$

$$\alpha_{i,n+2} = \begin{cases} 1 \text{ if the value of } \hat{\mathcal{A}}_i \text{ for the sequence } \Delta_s, \\ \Delta_s, \ldots, \Delta_s, \ldots \text{ is null,} \\ 0 \text{ otherwise.} \end{cases}$$

The functions α_{ij} are variants of equivalent schemes, and conversely, if for schemes $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, \ldots, A_n)$ and $\mathcal{B}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$ $\alpha_{ij}(\mathcal{A}) = \alpha_{ij}(\mathcal{B})$ for $i = 0, 1, \ldots, n$ and $j = 1, 2, \ldots, n+1, n+2$, then $\mathcal{A} = \mathcal{B}$.

The canonical form of the scheme $\mathcal{A}$ is

$$\mathcal{A}^* \equiv \overline{\alpha}_{01} \llcorner \ldots \overline{\alpha}_{0n+2} \; {}_{0n+2}\llcorner \; \overline{01} \; \lrcorner \ldots \lrcorner_{n1} \; A_1 \overline{\alpha}_{11} \; \llcorner_{11} \ldots \overline{\alpha}_{1n+2}$$

$$\lrcorner_{02} \ldots \lrcorner_{n2} \; A_2 \ldots A_n \overline{\alpha} \; \llcorner_{n1} \; \llcorner_{n1} \ldots \overline{\alpha}_{nn+2} \; \llcorner_{nn+2} \; 0\overline{n+1}\lrcorner \ldots \lrcorner_{nn+1}$$

$$\lrcorner_{\eta} \; 0 \; \llcorner_{\eta} \; 0\overline{n+2}\lrcorner \ldots \lrcorner_{nn+2}.$$

The process of evaluating a linear scheme $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$ reveals not only the order in which the operators are to be applied but also that in which the logical conditions are to be satisfied. Yet the value of the scheme, for a given assignment, does not include this information about the logical conditions. The information is superfluous as far as knowledge of the value of the schemes is concerned. Accordingly, Yanov gives an alternative representation of the logical schemes of algorithms in terms of matrices--rectangular schemes that do not incorporate this feature.

He again starts with logical variables $p_1, p_2, \ldots, p_k$ and operators $A_1, A_2, \ldots, A_n$. Let α_{ij} ($i = 0, 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$) be any logical functions such that $\alpha_{ij}(\Delta_s) = 1$ and $\alpha_{im}(\Delta_s) = 0$ for $m \neq j$ if application of

operator A_j for assignment Δ_s is to follow application of operator A_i. This condition is satisfied, for example, by the functions α_{ij} introduced above for linear schemes.

The process of operation of a matrix is again defined inductively. Suppose we are given a matrix

$$\mathcal{A}(p_1, p_2, \ldots, p_k) \equiv \begin{array}{c|ccc} & A_1, A_2, \ldots, A_n & & \\ \hline A_0 & \alpha_{01} & \alpha_{02} & \alpha_{0n} \\ A_1 & & & \\ A_n & \alpha_{n1} & \alpha_{n2} & \alpha_{nn} \end{array},$$

where A_0 stands for an empty "start" operator, and an assignment $\Delta_{s_1}, \Delta_{s_2}, \ldots, \Delta_{s_m}, \ldots$

Step 1 Examine the values of $\alpha_{0_j}(\Delta_{s_1})$ and write down some A_{i_1} for which $\alpha_{0i_1}(\Delta_{s_1}) = 1$.

Suppose that m steps have been performed and that at step m operator A_{i_m} has been written down.

Step m + 1 Examine the i_m-th row of the matrix and write down an operator $A_{i_{m+1}}$ for which $\alpha_{i_m i_{m+1}}(\Delta_{s_{m+1}}) = 1$. The procedure terminates if all members of the row under examination are 0. The sequence $A_{i_1}, A_{i_2}, \ldots, A_{i_m}, \ldots$ of operators written down is a value of the matrix for the assignment $\Delta_{s_1}, \Delta_{s_2}, \ldots, \Delta_{s_m}, \ldots$

As in the case of linear schemes a shift distribution B delimits certain sequences of assignments as admissible

if among the values of $\mathcal{A}$ for this sequence there is one A_{i_1}, A_{i_2}, ..., A_{i_m}, ... such that for each m, $\Delta_{s_{m+1}} = \Delta_{s_m}$ or $\Delta_{s_{m+1}}$ differs from Δ_{s_m} in values of variables belonging to B_{i_m}. A matrix $\mathcal{A}(p_1, p_2, \ldots, p_k)$ defines a <u>matrix scheme</u> relative to a given shift distribution B if for each admissible sequence the associated value of $\mathcal{A}$ is unique.

The equivalence of matrix schemes is defined just as for linear schemes, and it is proved that the problem of deciding whether two schemes are equivalent with respect to a given shift distribution is solvable.

For matrix schemes, the problem of designating those transformations that are complete relative to equivalence has a nice solution. By a Π_φ transformation we mean a mapping such that $\varphi \to (\Pi_\varphi(\alpha) \equiv \alpha)$ is a tautology. By a Π^i_φ transformation of the matrix $\mathcal{A} \equiv \begin{array}{c|c} A_i & \alpha_{ij} \end{array}$ we mean the replacement of an arbitrary element α_{ij} of its i-th row with the function $\alpha'_{ij} = \Pi_\varphi(\alpha_{ij})$. Yanov proves that all $\Pi^i_{A_i''}$ transformations take matrix scheme $\mathcal{A}$ into equivalent scheme $\mathcal{B}$ and that any scheme $\mathcal{B}$ equivalent to given scheme $\mathcal{A}$ can be obtained from $\mathcal{A}$ by a $\Pi^i_{A_i''}$ transformation and the addition or deletion of zero rows or columns.

A few words about equivalent MS (matrix schemes) and LS (linear schemes) are in order. In this connection, some auxiliary concepts must be introduced. A matrix $\mathcal{A} \equiv \begin{array}{c|c} A_i & \alpha_{ij} \end{array}_m$ is called <u>complete</u> if for i = 0, 1, ..., n, $A_i'' \to \bigvee_{j=1}^{m} \alpha_{ij}$ is a tautology. A set $\{\alpha_i\}$ of logical functions is said to be <u>orthogonal</u> if $\alpha_i \cdot \alpha_j = 0$ for $i \neq j$. A matrix $\mathcal{A}$ is

orthogonal if each row is an orthogonal set. A complete orthogonal matrix is said to be normal. Of course, each orthogonal matrix is a matrix scheme for any shift distribution, and given an MS $\mathcal{A} \equiv \overline{A_i \mid \alpha_{ij}}$, one can obtain an orthogonal MS equivalent to $\mathcal{A}$, namely, $\mathcal{B} \equiv \overline{A_i \mid (A_i'' \cdot \alpha_{ij})}$; $\mathcal{B}$ is said to be a reduced form of $\mathcal{A}$.

To establish a correspondence between linear and matrix schemes, it is necessary to add to the columns of a matrix two marked "( )" and "." designating the empty period and the termination of the procedure, respectively. Given MS $\mathcal{A} \equiv \overline{A_i \mid \alpha_{ij}}$, note that the matrix

		A_1	A_2	...	A_n	.
	A_0	α_{01}	α_{02}		α_{0n}	α_{0n+1}
	A_1	α_{11}	α_{12}		α_{1n}	α_{1n+1}
(*)	⋮					
	⋮					
	A_n	α_{n1}	α_{n2}		α_{nn}	α_{nn+1}

where $\alpha_{in+1} \equiv \bigwedge_{j=1}^{n} \overline{\alpha}_{ij}$, is complete and equivalent to $\mathcal{A}$ for any shift distribution. Note also that a reduced complete MS is normal. As the reader doubtlessly anticipates, MS $\mathcal{A} \equiv \overline{A_i \mid \alpha_{ij}}$ and LS $\mathcal{A}(p_1, p_2, \ldots, p_k, A_1, A_2, \ldots, A_n)$ are equivalent for a given shift distribution if for all admissible sequences of assignments their values are coincident.

We recall now the functions α_{ij} and schemes $\hat{a}_i$ defined earlier in connection with the derivation of the canonical form of a linear scheme which came up in discussing Yanov's completeness proof. If we replace the schemes $\hat{a}_i$ by the schemes $a'_i \equiv 0 \llcorner_i a(A_1, \ldots, A_i \lrcorner_i, \ldots, A_n)$, and the functions α'_{ij} are defined, related to a'_i as the α_{ij} are to $\hat{a}_i$, it is not hard to verify that the scheme $\overline{A_i \mid \alpha'_{ij}}$ will be equivalent to the scheme a for any shift distribution.

Contrariwise, suppose that a matrix of the form (*) is given, and that for a given shift distribution, this matrix is a complete MS. The linear scheme

$$\overline{\alpha}_{01} \underset{01}{\llcorner} \cdots \overline{\alpha}_{0n+2} \underset{0n+2}{\llcorner} \underset{01}{\lrcorner} \cdots \underset{n1}{\lrcorner} A_1 \overline{\alpha}_{11} \underset{11}{\llcorner} \cdots \overline{\alpha}_{1n+2}$$
$$\llcorner \underset{1n+2}{} \lrcorner \underset{02}{} \cdots \underset{n2}{\lrcorner} A_2 \cdots A_n \overline{\alpha}_{n1} \underset{n1}{\llcorner} \cdots \overline{\alpha}_{nn+2} \llcorner \underset{nn+2}{} \lrcorner \underset{0n+1}{} \cdots$$
$$\lrcorner \underset{nn+1}{} \lrcorner 0 \underset{\eta}{} \llcorner \underset{\eta}{} \lrcorner \underset{0n+2}{} \cdots \lrcorner \underset{nn+2}{}$$

is equivalent to the MS for the given shift distribution. For example, consider the following MS with the universal shift distribution

	A_1	A_2	( )	·
A_0	P_1	$\overline{P}_1$	0	0
A_1	0	$\overline{P}_1$	0	P_1
A_2	$P_1 \cdot P_2$	$\overline{P}_1 \cdot P_2$	0	P_2

The scheme $\mathcal{A}$ equivalent to it is

$$\mathcal{A} \equiv \underset{1}{p_1}\underset{2}{\llcorner}\underset{1}{p_1}\underset{5}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}\underset{3}{A_1p_1}\underset{4}{\llcorner}\underset{2}{\bar{p}_1}\underset{6}{\llcorner}\underset{3}{\lrcorner}\ \underset{}{\lrcorner}\ \underset{}{\lrcorner}\underset{5}{A_2\bar{p}_1 \vee \bar{p}_2}\underset{6}{\llcorner}\underset{7}{p_1 \vee \bar{p}_2}\underset{7}{\llcorner}\underset{4}{p_2}\underset{}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner},$$

where the left brackets have been indexed consecutively from the left, and all occurrences of the function 1 have been deleted.

This scheme can be transformed:

$$\mathcal{A} = \underset{2}{p_1}\underset{5}{\llcorner}\underset{}{\lrcorner}\underset{2}{A_1\bar{p}_1}\underset{2}{\llcorner}\underset{6}{\lrcorner}\ \underset{}{\lrcorner}\underset{7}{A_2p_2}\underset{5}{\llcorner}\underset{7}{\bar{p}_1}\underset{8}{\llcorner}\underset{6}{\lrcorner}\underset{8}{p_2}\underset{4}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}$$

$$= \underset{5}{\lrcorner}\underset{2}{p_1}\underset{4}{\llcorner}\underset{2}{A_1\bar{p}_1}\underset{1}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}\underset{7}{A_2p_2}\underset{5}{\llcorner}\underset{6}{\bar{p}_1}\underset{7}{\llcorner}\underset{4}{p_1}\underset{}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}$$

$$= \underset{5}{\lrcorner}\ \underset{6}{\lrcorner}\underset{2}{p_1}\underset{4}{\llcorner}\underset{2}{A_1\bar{p}_1}\underset{}{\llcorner}\underset{}{\lrcorner}\underset{7}{A_2p_2}\underset{5}{\llcorner}\underset{6}{\bar{p}_1}\underset{7}{\llcorner}\underset{4}{0}\underset{}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}$$

$$= \underset{6}{\lrcorner}\underset{2}{p_1}\underset{4}{\llcorner}\underset{2}{A_1\bar{p}_1}\underset{}{\llcorner}\underset{}{\lrcorner}\underset{7}{A_2p_2}\underset{5}{\llcorner}\underset{5}{\bar{p}_1}\underset{6}{\llcorner}\underset{7}{0}\underset{4}{\llcorner}\underset{}{\lrcorner}\ \underset{}{\lrcorner}$$

$$= \underset{6}{\lrcorner}\underset{}{p_1A_1\bar{p}_1}\underset{4}{\llcorner}\underset{2}{\llcorner}\underset{6}{\bar{p}_2}\underset{4}{\llcorner}\underset{}{\lrcorner}.$$

In a computer program, for various reasons, the same group of instructions may appear at different locations of the program. To reflect this situation, it is natural to consider logical schemes that contain repetitions of the operators. This can be denoted by retaining the same symbolism and adding a relation of the form $A_1 = A_2$. More generally, let there be given linear schemes $\mathcal{A}(p_1, p_2, \ldots,$

$p_k, A_1, A_2, \ldots, A_n)$ and $\mathcal{B}(p_1, \ldots, p_k, A_1, \ldots, A_n)$, together with t defining relations $R_1(A_{i_1}', A_{i_2}', \ldots, A_{i_{n_1}}')$, $R_2(A_{i_1}^2, A_{i_2}^2, \ldots, A_{i_{n_2}}^2), \ldots, R_t(A_{i_1}^t, A_{i_2}^t, \ldots, A_{i_{n_t}}^t)$, and consider the problem of the equivalence of $\mathcal{A}$ and $\mathcal{B}$. In this generality, however, all is in vain since the problem includes the word problem for semigroups, which, as is well known, is recursively unsolvable. More special cases can, however, be studied. Yanov does consider the case in which the lone defining relation is $A_1 = A_2$ and proves that the associated equivalence problem is solvable.

Yanov's paper constitutes an impressive, finished piece of work. The technical level is high, and all questions considered are nailed down very tightly. Within the limits he sets up, the logical domain is thoroughly covered, and such care and completeness are to be welcomed. It should be pointed out, nevertheless, that the territory covered invites and facilitates such a treatment. The mathematical problems are easily identified and quite "doable."

The definition of equivalence employed is surely too strong to be of much practical value. It seems right that two computer programs could be judged to be very alike, and hence "equivalent," without prescribing the same sequence of computations for all allowable inputs in some well-defined context. They could conceivably be required to agree only in the significant or "major" intermediate calculations. Of course, this is the delicate point. What and how much is to

be regarded as major? How much can things be disturbed in various directions without making the problem unsolvable?

Rutledge [31] extends some of Yanov's notions, and casts the theory in the guise of finite automata; that is, a scheme becomes a sextuple consisting of sets of operators, predicates, state functions, and the like. It represents an abstraction from the concrete peculiarities of Yanov's schemes. Rutledge also solves the equivalence problem by making use of canonical forms. He shows how to associate with each scheme a finite automaton, and by having his output function be a strictly into mapping of the set A of operators into itself, he arrives at a solution of the equivalence problem for the case in which certain operators are identified.

On the other hand, Rutledge's format is rather skeletal compared to Yanov's. It is certainly of value in characterizing exactly the mathematics involved, that is, as a part of the theory of finite automata, but it leaves an inexperienced reader with a feeling of incompleteness. The schemes of Yanov are symbolically more similar to computer programs, and hence are more suitable for performing the calculations that mirror those which come up in actual programs. Yanov's formalism is more appropriate for investigating the logical transformations performed on schemes; whereas Rutledge's serves to bring out the central organizing principles.

2.2.2. Nonlogical Transformations

2.2.2.1. Transformations on the Indices of Operators.

The question underlying the developments of this section concerns the description of the data to be fed into a computer. Changes in the initial description may be occasioned both by the programmed algorithm and by considerations of machine storage. To elaborate, consider an initial indexing, or parametrization of the elements in the domain of operators of some program scheme. Here the elements are indexed as independent units of information. The algorithm described by the scheme gives rise to a second indexing in which certain relations of dependence are expressed among the original parameters. This second parametrization leads to a new program scheme. Finally, the storage of the data of the problem occasions a third, operating parametrization, and this in turn gives rise to the operating program scheme, on the basis of which the actual program is to be constructed. The obvious aim is the development of procedures such that, given scheme $\mathcal{A}_1$ based on parametrization P_1 and a new parametrization P_2, one can construct from $\mathcal{A}_1$ a scheme $\mathcal{A}_2$ based on P_2. In other words, how do the representations of the arithmetical, control, and logical operations in a program scheme change when a new parametrization is introduced? This question is taken up by R. I. Podlovchenko [28] and N. G. Arsenteva [2]. These authors actually do little more, however, than provide illustrative examples. Let us consider the construction of the program scheme describing

the product A · B of symmetric n x n matrices A and B. In this case, the symmetry condition will affect the parametrization, and hence the operating program scheme.

One starts with the calculation scheme

$$\prod_{\ell=1}^{n}\prod_{i=1}^{n}\prod_{j=1}^{n} A_{ij\ell}\ .$$

A program scheme based on this calculation scheme is

$$1,2,3 \quad \downarrow \quad A_{ij\ell}F(j)f(j)p(j > n)\uparrow^{1}F(-nj)F(i)f(i)(1 \rightarrow j)$$

$$p(i > n)\uparrow^{2}(1 \rightarrow i)F(-ni)F(\ell)f(\ell)p(\ell > n)\uparrow^{3}F(-n\ell)$$

$$(1 \rightarrow \ell)\ \text{STOP},$$

where F, as above, is a readdressing operator; f(i) symbolizes that 1 is added to contents of the location storing i; and "a → b" means that a is stored in the location. containing b, that is, a displaces b in this location. Initially, i = j = ℓ = 1. If $A = (a_{ij})$, $B = (b_{k\ell})$, $C = A\cdot B = (c_{mr})$, and A and B are symmetric, new parameters i', j', k', ℓ', m', r' are introduced

$$i = p(i \geq j)\uparrow^{1}i'\downarrow_{1}j', \qquad \ell = p(k \geq \ell)\uparrow^{1}\ell'\downarrow_{1}k',$$

$$j = p(i \geq j)\uparrow^{1}j'\downarrow_{1}i', \qquad m = p(m \geq r)\uparrow^{1}m'\downarrow_{1}r',$$

$$k = p(k \geq \ell)\uparrow^{1}k'\downarrow_{1}\ell', \qquad r = p(m \geq r)\uparrow^{1}r'\downarrow_{1}m'.$$

Thus, the first of these equations is to be interpreted as stating that $i = i'$ when $i \geq j$ and that otherwise $i = j'$.

The following relations hold between the address-modification operators:

$$(3)\quad \begin{cases} F(i) \equiv p(i \geq j)\uparrow_1 F(i')\uparrow_2\downarrow_1 F(j')\downarrow_2; & F(k) \equiv p(k \geq \iota)\uparrow_1 F(k')\uparrow_2\downarrow_1 F(\iota')\downarrow_2, \\ F(j) \equiv p(i > j)\uparrow_1 F(j)\uparrow_2\downarrow_1 F(i')\downarrow_2; & F(\iota) \equiv p(k > \iota)\uparrow_1 F(\iota')\uparrow_2\downarrow_1 F(k')\downarrow_2, \\ F(m) \equiv p(m \geq r)\uparrow_1 F(m')\uparrow_2\downarrow_1 F(r')\downarrow_2; & F(r) = p(m > r)\uparrow_1 F(r')\uparrow_2\downarrow_1 F(m')\downarrow_2, \end{cases}$$

and similarly replacing i, j, k, ι, m, r, i', j', k', ι', m', r' by $-i$, $-j$, $-k$, $-\iota$, $-m$, $-r$, $-i'$, $-j'$, $-k'$, $-\iota'$, $-m'$, and $-r'$, respectively, as the argument of the operator F.

It is economical to introduce "formation operators" which store the initial form of an instruction. The notation "$[1 \to j]$" denotes the operator that sets $j = 1$ in operators depending on j, and replaces j in its location by 1.

Noting that in the case under consideration $k = j$, $m = i$, $r = \iota$, system (3) is replaced by

$$(4)\quad \begin{cases} F(j) = p(i > j)\uparrow_1 F(j')\uparrow_2\downarrow_1 F(i')\downarrow_2 p(j \geq \iota)\uparrow_3 F(k')\uparrow_4\downarrow_3 F(\iota')\downarrow_4, \\ F(i) = p(i \geq j)\uparrow_1 F(i')\uparrow_2\downarrow_1 F(j')\downarrow_2 p(i \geq k)\uparrow_3 F(m')\uparrow_4\downarrow_3 F(r')\downarrow_4, \\ F(\iota) = p(j > \iota)\uparrow_1 F(\iota')\uparrow_2\downarrow_1 F(k')\downarrow_2 p(i > \iota)\uparrow_3 F(r')\uparrow_4\downarrow_3 F(m')\downarrow_4. \end{cases}$$

Using (4), formation operators, and the fact that some logical conditions are universally true and others are universally false, program scheme (1) becomes

$$\underset{1,2,3}{\downarrow} A_{ij\ell}\, p(i > j)\overset{4}{\uparrow}F(j')\overset{5}{\uparrow\downarrow}F(i')\overset{4}{\downarrow}p(j \geq \ell)\overset{6}{\uparrow}F(k')\overset{7}{\uparrow\downarrow}F(\ell')\overset{6}{\downarrow}f(j)\overset{7}{\downarrow}$$

$$p(j > n)\overset{1}{\uparrow}[1 \rightarrow j]F(i')p(i \geq \ell)\overset{8}{\uparrow}F(m')\overset{8}{\uparrow\downarrow}F(r')\overset{9}{\downarrow}f(i)p(i > n)$$

$$\overset{2}{\uparrow}[1 \rightarrow i]F(k')F(m')f(\ell)p(\ell > n)\overset{3}{\uparrow}[1 \rightarrow \ell]\ \text{STOP}.$$

It is assumed that the elements of the matrices are stored by rows into successive memory locations, starting with a_{11} in a_1, b_{11} in b_1, and c_{11} in c_1. Memory parameters s, t, v are introduced to represent the fact that the elements $a_{i'j'}$, $b_{k',\ell'}$, $c_{m',r'}$ are stored, respectively, in locations

$$a_s = a_1 + (j' - 1) + \frac{i'(i' - 1)}{2},$$

$$b_t = b_1 + (\ell' - 1) + \frac{k'(k' - 1)}{2},$$

$$c_v = c_1 + (r' - 1) + \frac{m'(m' - 1)}{2}.$$

Making use of the fact that in the case under consideration $k = j$, $m = i$, $r = \ell$, it follows that

$$i' = p(i \geq j)\overset{1}{\uparrow}i\overset{1}{\downarrow}j,$$

$$k' = p(j \geq \ell)\overset{1}{\uparrow}j\overset{1}{\downarrow}\ell \ ,$$

$$m' = p(i \geq \ell)\overset{1}{\uparrow}i\overset{1}{\downarrow}\ell \ .$$

Consequently,

$$(5)\begin{cases} F(i') \equiv p(i \geq j)\overset{1}{\uparrow}F(is)\overset{2}{\uparrow}\overset{1}{\downarrow}F(js)\overset{2}{\downarrow} \ , \\ F(j') \equiv p(j \geq \ell)\overset{1}{\uparrow}F(jt)\overset{2}{\uparrow}\overset{1}{\downarrow}F(\ell t)\overset{2}{\downarrow} \ , \\ F(m') \equiv p(i \geq \ell)\overset{1}{\uparrow}F(iv)\overset{2}{\uparrow}\overset{1}{\downarrow}F(\ell v)\overset{2}{\downarrow} \ . \end{cases}$$

Substituting in (4), one obtains

$$F(j) \equiv p(i > j)\overset{1}{\uparrow}F(s)\overset{2}{\uparrow}\overset{1}{\downarrow}F(js)\overset{2}{\downarrow}p(j \geq \ell)\overset{3}{\uparrow}F(jt)\overset{4}{\uparrow}\overset{3}{\downarrow}F(t)\overset{4}{\downarrow},$$

$$F(i) \equiv p(i \geq j)\overset{1}{\uparrow}F(is)\overset{2}{\uparrow}\overset{1}{\downarrow}F(s)\overset{2}{\downarrow}p(i \geq \ell)\overset{3}{\uparrow}F(iv)\overset{4}{\uparrow}\overset{3}{\downarrow}F(v)\overset{4}{\downarrow},$$

$$F(\ell) \equiv p(j > \ell)\overset{1}{\uparrow}F(t)\overset{2}{\uparrow}\overset{1}{\downarrow}F(\ell t)\overset{2}{\downarrow}p(i > \ell)\overset{3}{\uparrow}F(v)\overset{4}{\uparrow}\overset{3}{\downarrow}F(\ell v)\overset{4}{\downarrow}.$$

One finally arrives at the following operating scheme, based on the operating parametrization, and employing the memory parameters:

$$\underset{1,2,3}{\downarrow} \ A_{stv}p(i > j)\overset{4}{\uparrow}F(s)\overset{5}{\uparrow}\overset{4}{\downarrow}F(js)\overset{5}{\downarrow}p(j \geq \ell)\overset{6}{\uparrow}F(jt)\overset{7}{\uparrow}$$

$$\underset{6}{\downarrow}F(t)\underset{7}{\downarrow}f(j)p(j > n)\overset{1}{\uparrow}[1 \rightarrow j]F(is)p(i \geq \imath)\overset{8}{\uparrow}F(iv)\overset{9}{\uparrow}\downarrow$$

$$F(v)\underset{9}{\downarrow}f(i)p(i > n)\overset{2}{\uparrow}[1 \rightarrow i]F(\imath t)F(\imath v)f(\imath)p(\imath > n)\overset{3}{\uparrow}[1 \rightarrow \imath]\ \text{STOP}.$$

Other examples are given in the two cited papers: The Leverrier method of finding the characteristic polynomial of a matrix, the solution of the Dirichlet problem over a square, and others.

It must be clear to the reader that the tone, style, and methods of these papers are of a piece with those of Lyapunov's earlier paper. The symbolism and methods of the papers are purely descriptive, and they are written in a way that looks forward to a more general, systematic analysis of the problem. Podlovchenko's, On the Basic Principles of Programming, I [27], seemed to forecast such a treatment, and in that sense his [28] is disappointing. His [27] is concerned with the introduction of definitions and terminology that serve to organize the data of programming into classes of information, and with the systematic identification of this information. Chief among the notions introduced are those of the characteristic of a system of information, the characteristic of an operator, and a ranking of operators. The characteristic of a system of information is the set of parameters indexing the elements of the set. In the above example, A, B, and C constitute a system of information and {i, j, k, l, m, r} is the characteristic of this system, with {i, j} being the characteristic

of A, etc. When restrictions are placed on some of the parameters of a parameterized system of information we arrive at a subcharacteristic of this system; {i, j, k} is a subcharacteristic of A, B, C. The argument and value of an operator symbolizing some operation on an information class are called its input and output, respectively. A subcharacteristic of the input determines a characteristic of the operator; in the above {i, j, ℓ} is a characteristic of operator $A_{ij\ell}$.

Suppose we have an operator $A_{i_1 \ldots i_n}$ with characteristic $\{i_1, \ldots, i_n\}$ and i_1 assumes some one of s allowable values. The resulting operator depends on $\{i_2, \ldots, i_n\}$ and there are s such operators. The logical product $B_{i_2 i_3 \ldots i_n}$ of these s operators is said to be an operator of the first rank relative to $A_{i_1 \ldots i_n}$. Operators of higher rank relative to $A_{i_1 \ldots i_n}$ are defined recursively. Thus, an operator of the n-th rank relative to $A_{i_1 \ldots i_n}$ is independent of parameters.

In short, while [27] seems to make a beginning for a formal treatment of transformations on indices in schemes, [28], for the most part, merely catalogues examples.

2.2.2.2. An Example. The transformations of program schemes by means of changes in the indices of the operators bear on several special classes of problems, such as those in which some iterative procedure is applied in some definite order to each of the arguments of some function of

several variables, and those which involve branching processes. An illustration of the former is the calculation of the partial derivative of a function $f(x_1, \ldots, x_p)$ of p variables given values $x_1^{(1)}, x_1^{(2)}, \ldots, x^{(\nu_1)}, x_2^{(1)}, \ldots x_2^{(\nu_2)}, \ldots, x_p^{(1)}, \ldots x_p^{(\nu_p)}$ of the arguments and hence $\nu_1 \cdot \nu_2 \cdots \nu_p$ function values. Suppose we wish to compute (approximately)

$$\frac{\partial(\bar{x})}{\partial xp}, \text{ where } \bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_p) \text{ and that } x_k^{(m_k)} < x_k^{(m_k+1)},$$

and there is a positive integer N such that $m_k + N \leq \nu_k$, $k = 1, 2, \ldots, p$. The value of f at $\bar{x}$ is computed by successive polynomial interpolation, and thence the partial derivative of f with respect to x_p at $\bar{x}$ is calculated. Suppose it is desired to compute the partial derivative of f at $\bar{x}$ with respect to each of the other coordinates. A calculation scheme can be written sketching the procedure for calculating the partial derivatives of f at $\bar{x}$; if these schemes are arranged in some definite order, then each can be obtained from the preceding one by a "circular substitution of the subscripts." The objective is the construction of a program scheme that includes the algorithm prescribed by the succession of calculation schemes. To illustrate, let $p = 3$, and let $A_{i_3 i_2 i_1}$ be some operator such that the memory cell on which $A_{i_3 i_2 i_1}$ acts depends on i_3, i_2, i_1 according to some rule $\phi(i_3, i_2, i_1)$. For instance, we might have

$$\phi(i_3, i_2, i_1) = r_3 i_3 + r_2 i_2 + r_1 i_1,$$

where r_1, r_2, and r_3 are positive integers. Thus, if we start at some memory cell α, $A_{i_3 i_2 i_1}$ is applied to the contents of the cell $\alpha + r_3 i_3 + r_2 i_2 + r_1 i_1$.

We start from the schemes

$$(1)\quad \prod_{i_3=m_3}^{n_3-1} \left\{ \prod_{i_2=m_2}^{n_2-1} \left\{ \prod_{i_1=m_1}^{n_1-1} \left\{ A_{i_3 i_2 i_1} \right\} \right\} \right\},$$

$$(2)\quad \prod_{i_1=m_1}^{n_1-1} \left\{ \prod_{i_3=m_3}^{n_3-1} \left\{ \prod_{i_2=m_2}^{n_2-1} \left\{ A_{i_3 i_2 i_1} \right\} \right\} \right\},$$

$$(3)\quad \prod_{i_2=m_2}^{n_2-1} \left\{ \prod_{i_1=m_1}^{n_1-1} \left\{ \prod_{i_3=m_3}^{n_3-1} \left\{ A_{i_3 i_2 i_1} \right\} \right\} \right\}.$$

Scheme (1) gives rise to the scheme

$$(4)\begin{cases} (A_{m_3 m_2 m_1} \to A_{i_3 i_2 i_1})(m_3 \to \delta_3)\overset{3}{\downarrow}(m_2 \to \delta_2)\overset{2}{\downarrow} \\ (m_1 \to \delta_1)\overset{1}{\downarrow} A_{i_3 i_2 i_1} F(1,i_1) f(1,\delta_1) \mu(\bar{\delta} > n_1)\overset{1}{\uparrow} F(-(n_1-m_1),i_1) \\ F(1, i_2) f(1, \delta_2) \mu(\bar{\delta}_2 > n_2)\overset{2}{\uparrow} F(-(n_2-m_2), i_2) \\ F(1, i_3) f(1, \delta_3) \mu(\bar{\delta}_3 > n_3)\overset{3}{\uparrow} \text{ STOP}. \end{cases}$$

Here $(A_m \to A_i)$ means that operator A_i is replaced by operator A_m; $f(1, \delta)$ means that the contents of cell δ are

increased by one; and F(m, i) means that m is added to the parameter i. Also $\overline{\delta}$ designates the contents of the cell δ.

The memory parameter t is introduced via the equations

$$\alpha + \phi(i_3, i_2, i_1) = \alpha + t,$$
$$\alpha + \phi(m_3, m_2, m_1) = \alpha + t_0,$$

where $\phi(i_3, i_2, i_1) = r_3 i_3 + r_2 i_2 + r_1 i_1$.

We write $\overline{A}_t$ for $A_{i_3 i_2 i_1}$ and $\overline{A}_{t_0}$ for $A_{m_3 m_2 m_1}$.

Taking into account the changes occasioned in the address substitution operators, the above scheme is transformed into

$$(\overline{A}_{t_1} \rightarrow \overline{A}_t)(m_3 \rightarrow \delta_3)\overset{3}{\downarrow}(m_2 \rightarrow \delta_2)\overset{2}{\downarrow}(m_1 \rightarrow \delta_1)\overset{1}{\downarrow}\overline{A}_t$$

$$F(r_1, t)f(1,\delta_1)\mu(\overline{\delta}_1 > n_1)\overset{1}{\uparrow}F(-(n_1 - m_1)r_1, t)F(r_2, t)f(1, \delta_2)$$

$$\mu(\overline{\delta}_2 > n_2)\overset{2}{\uparrow}F(-(n_2 - m_2)r_2, t)F(r_3, t)f(1,\delta_3)\mu(\overline{\delta}_3 > n_3)\overset{3}{\uparrow} \text{ STOP.}$$

To achieve symmetry of representation, zero is represented by each of n_0, m_0, r_0, and successive address substitution operators are combined to arrive at:

$$(\overline{A}_{t_0} \rightarrow \overline{A}_t)(m_3 \rightarrow \delta_3)\overset{3}{\downarrow}(m_2 \rightarrow \delta_2)\overset{2}{\downarrow}(m_1 \rightarrow \delta_1)\overset{1}{\downarrow}\overline{A}_t$$

$$F(r_1 - r_0(n_0 - m_0), t)f(1, \delta_1)\mu(\overline{\delta}_1 > n_1)\overset{1}{\uparrow}F(r_2 - r_1(n_1 - m_1), t)$$

$$f(1,\delta_2)\mu(\overline{\delta}_2 > n_2)\overset{2}{\uparrow}F(r_3 - r_2(n_2 - m_2), t)f(1,\delta_3)\mu(\overline{\delta}_3 > n_3)\overset{3}{\uparrow} \text{ STOP.}$$

Let $\phi_k = r_k - r_{k-1}(n_{k-1} - m_{k-1})$. This enables one to write

$$(\overline{A}_{t_0} \to A_t) \prod_{s=0}^{3} \{(m_s \to \delta_s)\} \overset{0}{\downarrow} \overline{A}_t F(a_1,t) \prod_{\iota=0}^{0} \{m_\iota \to \delta_\iota)\}$$

$$f(1,\delta_1)\mu(\overline{\delta}_1 > n_1) \overset{0}{\uparrow} F(a_2,t) \prod_{\iota=0}^{1} \{(m_\iota \to \delta_\iota)\} f(1,\delta_2)\mu(\overline{\delta}_2 > n_2) \overset{0}{\uparrow}$$

$$F(a_3,t) \prod_{\iota=0}^{2} \{(m_\iota \to \delta_\iota)\} f(1,\delta_3)\mu(\overline{\delta}_3 > n_3) \overset{0}{\uparrow} \text{ STOP.}$$

It is to be understood that the numbers a_k are stored in cells α_k, the m_k in β'_k, and the n_k in β''_k. We put

$$\Phi_k = F(\overline{\alpha}_k,t) \prod_{\iota=0}^{k-1} \{(\overline{\beta}'_\iota \to \delta_\iota)\} f(1,\delta_k)\mu(\overline{\delta}_k > \overline{\beta}''_k) \overset{0}{\uparrow}.$$

With this abbreviation, the above scheme becomes

$$(*) \quad (\overline{A}_{t_0} \to \overline{A}_t) \prod_{s=1}^{3} \{(\overline{\beta}'_s \to \delta_s)\} \overset{0}{\downarrow} \overline{A}_t \Phi_1 \Phi_2 \Phi_3 \text{ STOP.}$$

By expansion of the products, this can be rearranged to become

$$(\overline{A}_{t_0} \to \overline{A}_t) \prod_{s=1}^{3} \{(\overline{\beta}'_s \to \delta_s)\} \overset{0}{\downarrow} \overline{A}_t (1 \to \gamma_k)(\Phi_1 \to \Phi_k) \overset{1}{\downarrow} \Phi_k$$

$$F(1,k)f(1,\gamma_k)\mu(\overline{\gamma}_k > 3) \overset{1}{\uparrow} \text{ STOP,}$$

where the operator $(\Phi_1 \to \Phi_k)$ is construed as the execution of the following three operators:

$$\{F(\overline{\alpha}_1,t) \to F(\overline{\alpha}_k,t)\},$$

$$\{f(1,\delta_1) \to f(1,\delta_k)\},$$

$$\{\mu(\overline{\delta}_1 > n_1) \overset{0}{\uparrow} \to \mu(\overline{\delta}_k > n_k) \overset{0}{\uparrow}\}.$$

The consecutive execution of calculation schemes (1), (2), and (3) then is prescribed by the following program scheme:

$$\prod_{\nu=0}^{3} \{(a_\nu \to \alpha_\nu)(m_\nu \to \beta'_\nu)(n_\nu \to \beta''_\nu)\} \prod_{y=1}^{3} \{(\overline{\alpha}_y \to \theta_\alpha)(\overline{\beta}'_y \to \theta_{\beta'})$$

$$(\overline{\beta}''_y \to \theta_{\beta''}) \prod_{i=3}^{2} \{(\overline{\alpha}_{\tau-1} \to \alpha_\tau)(\overline{\beta}_{\tau-1} \to \beta'_\tau)(\overline{\beta}''_{\tau-1} \to \beta''_\tau)$$

$$\overline{\theta}_\alpha \to \alpha_1)(\overline{\theta}_{\beta'} \to \beta'_1)(\theta_{\beta''} \to \beta''_1)\mathcal{A}^{(3)}\} \text{ STOP},$$

where θ_α, $\theta_{\beta'}$, and $\theta_{\beta''}$ are "working cells," that is, cells in which intermediate calculations are carried out and $\mathcal{A}^{(3)}$ stands for the programming scheme (*).

2.3. An Abstract Formulation

R. I. Podlovchenko's "On the Transformations of Program Schemes and Their Applications to Programming" [29], seems to mark the first abstract treatment of the Lyapunov operator method of programming. It is wise to consider it separately since its subject is the systematization and abstract treatment of the notions introduced in the more concrete, special instances described above. Other papers on which we shall comment are conveniently suited to a more common treatment in that they introduce or develop notions absent or scarcely present in the more practical papers.

This paper occupies a pivotal position in the development of the theory. It presents a connected series of concepts which represent an abstraction from the notions introduced in the earlier, problem-oriented papers, but which, at the same time, are directly and easily reducible to these notions. The paper thus serves to unify the contents of the papers described previously, and provides the reader with a rigorous, convenient source of the basic topics scattered among them.

On the other hand, the possibility of defining a fruitful algebraic structure on the classes of objects considered by Podlovchenko is not pursued. The theoretical development remains therefore on the level of mere systematization of and generalization from concrete instances of program schemes. The concepts are not related to those of any established branch of algebra, and hence the possibility

of pursuing, by analogy, suggested directions of research as well as use of the battery of techniques provided by such disciplines are foregone. Podlovchenko's paper thus appears as transitional, summing up the basic notions so far introduced, and keeping manageable the distance between abstract concept and particular example.

2.3.1. Basic Definitions. By a memory is meant a set $Z = \{z\}$ of objects whose members are called cells. Consider a set $G = \{x\}$ of states and a set Y of mappings called states of the memory, or memory states, of Z into G. An operator is a function with domain Y and range a subset of Y. By a predicate over the memory–states is meant a function p defined on Y and taking values in the set $\{0,1\}$. By the product (composition) $A_1 \cdot A_2$ of the operators A_1 and A_2 is meant the operator $A = A_2(A_1(X))$, $X \epsilon Y$. The operator

$$A(X) = \begin{cases} A_1(X) & \text{if } p_\eta(X) = 1, \\ A_2(X) & \text{if } p_\eta(X) = 0 \end{cases}$$

is called conditional.

If $\mathcal{A}$ is some set of operators and $\mathcal{B}$ is a set of predicates over the memory–states (hereafter for brevity the members of $\mathcal{B}$ are called simply predicates), then we consider the smallest superset $\mathcal{A}_\mathcal{Z}$ of $\mathcal{A}$ which is closed under composition and forming the conditional of two operators with respect to some predicate $p_n \subset \mathcal{B}$.

It is clear that the members of $\mathcal{A}_{\mathcal{B}}$ constitute the main ingredients out of which program schemes are made. Podlovchenko accordingly introduces his definition of a scheme with a view toward representing members of $\mathcal{A}_{\mathcal{B}}$.

By a <u>logical condition</u> is meant any member of the set $V_{\mathcal{B}} = \{p_\eta(X), \iota)\}$, where $p_\eta \in \mathcal{B}$ and ι is a natural number. Logical conditions $(p_{\eta_1}(X), \iota_1)$ and $(p_{\eta_2}(X), \iota_2)$ are said to be <u>equal</u> if $p_{\eta_1} \equiv p_{\eta_2}$ and $\iota_1 = \iota_2$. A function mapping $\mathcal{A}_{U V_{\mathcal{B}}}$ into itself is said to be a <u>formulator</u>. Let $\mathcal{F} = \{\Phi_p\}$ be some set of formulators fixed in the sequel. A functor is an element of the set $W_{\mathcal{F}} = \{(\Phi_p, \iota, m)\}$ of ordered triples, where $\Phi_p \in \mathcal{F}$, and ι and m are natural numbers. The definition of equality of functors is obvious. The interpretation of an occurrence of a functor (Φ, ι, m) in a scheme is as follows: ι defines that element of the scheme that serves as an argument of the function Φ, and the value of Φ for this argument is that element of the scheme defined by the number m.

Let $B(\tau, X)$ be a function from N x Y, where N is the set of natural numbers, into $\mathcal{A}_{U V_{\mathcal{B}} U W_{\mathcal{F}}}$ which for t = 0 is independent of X. Consider a sequence of such functions:

$$M^{n,s}(\tau,X) = B^0(\tau,X), B^1(\tau,X), \ldots, B^n(\tau,X), B^{n+1}(\tau.X), \ldots, B^{n+s}(\tau,X).$$

It is such a sequence, with suitable restrictions, that is to serve as a scheme. It is necessary to define how to make a sequence $M^{n,s}(\tau,x)$ "go," and in this connection to define its admissible applications. Given a pair (τ,X_0), then a triple $X(\tau,X_0)$, $k(\tau_0,X_0)$, $M^{n,s}(\tau_0,X_0)$, where $X(\tau_0,X_0)$ is a memory-state and $k(\tau_0,X_0)$ is a non-negative integer, is said to be admissible if

(a) $0 \leq k(\tau_0,X_0) \leq n$;

(b) $B^{k(\tau_0,X_0)}(\tau_0,X_0)$ is not a functor, or $B^{k(\tau_0,X_0)} = (\Phi,\ell,m)$, where $0 \leq \ell$, $m \leq n + s$, and $B^{\ell}(\tau_0,X_0) \notin W_{\mathcal{A}}$.

It is necessary to give the rule θ of succession for admissible triples. θ is defined by stating how to proceed from an admissible triple $X(\tau_0,X_0)$, $k(\tau_0,X_0)$, $M^{n,s}(\tau_0,X_0)$ to a next admissible triple $X(\tau_0 + 1,X_0)$, $k(\tau_0 + 1,X_0)$, $M^{n,s}(\tau_0 + 1,X_0)$, as follows:

$$\text{(a)}\quad X(\tau_0 + 1,X_0) = \begin{cases} A(X(\tau_0,X_0)) \text{ if } B^{k(\tau_0X_0)}(\tau_0,X_0) = A \in \mathcal{A}., \\ X(\tau_0,X_0) \text{ otherwise}; \end{cases}$$

$$\text{(b)}\quad k(\tau_0 + 1,X_0) = \begin{cases} \ell \text{ if } B^{k(\tau_0,X_0)}(\tau_0,X_0) = (p(X),\ell) \in V_{\mathcal{B}} \\ \text{and } p(X(\tau_0,X_0)) = 0, \\ k(\tau_0,X_0) \text{ otherwise}; \end{cases}$$

$$(c)\quad M^{n,s}(\tau_0+1,X_0) = \begin{cases} M^{n,s}(\tau_0,X_0)[\Phi(B^{\ell}(\tau_0,X_0) \to m] \text{ if} \\ \qquad B^{k(\tau_0,X_0)}(\tau_0,X_0) = [\Phi,\ell,m) \in W, \\ \qquad \blacksquare \\ M^{n,s}(\tau_0,X_0) \text{ otherwise,} \end{cases}$$

where $M^{n,s}(\tau_0,X_0)[\Phi(B^{\ell}(\tau_0,X_0)) \to m]$ is the sequence obtained from $M^{n,s}(\tau_0,X_0)$ by substitution of $\Phi(B^{\ell}(\tau_0,X_0)) \in A \cup V_{\mathcal{B}}$ for the m-th element of the sequence $M^{n,s}(\tau_0,X_0)$.

It is natural to call a triple <u>final</u>, or concluding, if application of θ to this triple results in an inadmissible triple. Thus $X(\tau_0,X_0)$, $k(\tau_0,X_0)$, $M^{n,s}(\tau_0,X_0)$ is final if

(a) $k(\tau_0,X_0) = n$, and $B^n(\tau_0,X_0)$ is an operator or functor

or

(b) $k(\tau_0,X_0) = n$, $B^n(\tau_0,X_0) = (p^n(X),\ell^k)$, and

$$p^n(X(\tau_0,X_0)) = 1;$$

or

(c) $0 \leq k(\tau_0,X_0) = k_0 \leq n$, $B^{k0}(\tau_0,X_0) = (p^{k0}(X),\ell^{k0})$, and

$$p^{k0}(X(\tau_0,X_0)) = 0, \ell^{k0} > n.$$

The function θ obviously successively generates triples, and to get started one sets $X(0,X_0) = X_0$ for $X_0 \in Y$ and $k(0,X_0) = 0$. Consider now a sequence of triples

$X(0,X_0)$	$X(1,X_t)$, ...,	$X(\nu,X_0)$, ...
$k(0,X_0)$	$k(1,X_0)$, ...,	$k(\nu,X_0)$, ...
$M^{n,s}(0,X_0)$	$M^{n,s}(1,X_0)$, ...,	$M^{n,s}(\nu,X_0)$, ...

A memory-state X_0 is said to be M-admissible if there is a concluding triple in (1), and a sequence $M^{n,s}(\tau,X)$ is said to be a scheme if there is at least one M-admissible $X_0 \in Y$. Thus, a sequence $M^{n,s}(\tau,X)$ is a scheme if for some X_0 the process of generating admissible triples comes to a halt.

Let $Y_M \subseteq Y$ be the set of all M-admissible memory-states. Toward representing the members of $\mathcal{A}_{\mathcal{B}}$, a few more auxiliary definitions are needed. Given $X_0 \in Y_M$, $\nu_M(X_0)$ is the number such that the triple $X(\nu_M(X_0), X_0), k(\nu_M(X_0), X_0), M^{n,s}(\nu_M(X_0), X_0)$ is concluding. Set $R_M(X_0) = X(\nu_M(X_0) + 1, X_0)$ and $L_M(X_0) = B^{k(0,X_0)}(0,X_0), B^{k(1,X_0)}(1,X_0), \ldots, B^{k(\nu_M(X_0), X_0)}(\nu_M(X_0), X_0)$. The scheme $M^{n,s}(\tau,X)$ is said to describe the operator A over the set Y_M if $A(X_0) = R_M(X_0)$ for $X_0 \in Y_M$. Thus, if $Y_M \neq Y$, a single scheme can describe different operators. Consider, now, the set M_0 of all schemes M such that

(a) the sequence $M(0,X_0)$ contains no functors;

(b) $Y_M = Y$;

(c) the set $\{L_M(X_0)\}$, $X_0 \in Y$, is finite, and each member terminates in an operator.

It follows that $\mathcal{A}_{\mathcal{B}} = \{R_M | M \in M_0\}$.

2.3.2. An Example. To illustrate some of the above notions, consider the sequence $M^{6,0} = A, B, (p_1,6), C, (p_1, 1), D, (p_2, 1)$. Here A, B, C, and D are operators, and p_1 and p_2 are predicates. We take the cases (a) $p_1 \equiv 1$,

$p_2 \equiv 1$, (b) $p_1 \equiv 0$, $p_2 \equiv 1$, (c) $p_1 \equiv 1$, $p_2 \equiv 0$, and (d) $p_1 \equiv 0$, $p_2 \equiv 0$. For an arbitrary initial $X_0 \in Y$, the sequences $X(\tau, X_0)$ and $k(\tau, X_0)$ in cases (a) and (b), respectively, are

(a)
$$\begin{array}{lllll} X_0 & A(X_0) & B(A(X_0)) & B(A(X_0)) & C(B(A(X_0))) \\ 0 & 1 & 2 & 3 & 4 \\ C(B(A(X_0))) & D(C(B(A(X_0)))) & D(C(B(A(X_0 & & \\ 5 & 6 & 7 & & \end{array}$$

(b)
$$\begin{array}{lllll} X_0 & A(X_0) & B(A(X_0)) & B(A(X_0)) & B(A(X_0)) \\ 0 & 1 & 2 & 6 & 7 \end{array}$$

Also, in case (a), $Y_{M6,0} = Y$, $L_M(X) = A, B, (p_1, 6), C, (p_1, 1), D, (p_2, 1)$, and $R_{M6,0} = A \cdot B \cdot C \cdot D$; whereas in case (b), $Y_{M6,0} = Y$, $L_M(X) = A, B, (p_1, 6), (p_2, 1)$, and $R_{M6,0} = A \cdot B$. In cases (c) and (d), the generated sequence of triples is infinite and contains no concluding triple. Hence, in these cases $Y_{M6,0} = \emptyset$ and $L_M(X)$ and $R_{M6,0}$ are undefined.

2.3.3. Subschemes. Prior to considering questions about the transformation of one scheme into another, basic questions on the relations of schemes arise. For example, when is a scheme M_2 to be considered a subscheme of scheme M_1? Also, what extra conditions must a scheme satisfy in order to serve as an obvious abstraction of Lyapunov's program schemes?

Given a scheme $M^{n,s}(\tau, X)$ and for $X_0 \in Y_M$ the sequence $L_M(X_0)$, the subsequence $L^*_M(X_0) = B^{k(p_0,X_0)}(p_0,X_0), \ldots,$

$B^k(p_\lambda, X_0)$ of $L_M(X_0)$ that consists exactly of the occurrences of operators in $L_M(X_0)$ is called an operator sequence. Here $X(p_i, X_0) = C^0 \cdot C^1 \ldots C^{i-1}(X_i)$, for $0 \leq i \leq \lambda$, where $C^m = B^{k(p_m, x_0)}(p_m, X_0)$, $m = 0, 1, \ldots, \lambda$. Suppose that $N_1(X_1^0) = C_1^0, \ldots, C_1^{n1}$ and $N_2(X_2^0) = C_2^0, C_2^1, \ldots, C_2^{n2}$, where X_1^0 and X_2^0 are memory-states, are two sequences of operators. Associated with each operator $C_\sigma^i (\sigma = 1, 2, i = 1, 2, \ldots, \mu_\sigma)$ is a memory-state $X_\sigma^i = C_\sigma^0 \cdot C_\sigma^1 \ldots C_\sigma^{i-1}(X^0)$. The sequence $N_2(X_2^0)$ is said to include the sequence $N_1(X_1^0)$ over the set $J \subseteq Z (N_2(X_2^0) \supseteq N_1(X_1^0)[J])$ if there is a subsequence $\tilde{N}_2(X_2^0) = C_2^{s0}, C_2^{s1}, \ldots, C_2^{sn1}$, of $N_2(X_2^0)$ such that

(a) $C_2^{s_i} = C_1^i$, $i = 0, 1, \ldots, n_1$,

(b) $X_2^{s_i}(Z) = X_1^i(Z)$ for $Z \in J$ and $i = 0, 1, \ldots, n_1$.

Let Y_{M_1} and Y_{M_2} be sets of admissible memory-states of schemes M_1 and M_2, respectively; then $X'' \in Y_{M_2}$ is said to be an M_1-associate of $X' \in Y_{M_1}$ over $J \subseteq Z$ if

(a) $L^*_{M_2}(X'') \supseteq L^*_{M_1}(X')[J]$,

and

(b) if over J, $R_{M_2}(X'') = R_{M_1}(X')$.

Consider a subset $Y^*_{M_1}$ of Y_{M_1}. Scheme M_2 is said to include scheme M_1 over sets J and Y_{M_1} if for each $X' \in Y^*_{M_1}$, there is at least one $X'' \in Y_{M_2}$ that is an M_1-associate of X' over J. Notation: $M_2 \overset{J}{\rightrightarrows} M_1[Y^*_{M_1}]$.

Thus, scheme M_1 is included in scheme M_2 relative to a subset T of the memory and subset $Y^*_{M_1}$ of the admissible states Y_{M_1} of M_1 if for each $X' \in Y_{M_1}$ there is an $X'' \in Y_{M_2}$ such that over T the operator sequence generated by M_2 includes the operator sequence generated by M_1 if the memory states at which M_1 and M_2 terminate coincide over T.

Podlovchenko makes some observations about this inclusion relation, and the simpler ones are as follows:

(1) If $T^* \subseteq T$ and $Y^{**}_{M_1} \subseteq Y^*_{M_1}$ and $M_2 \overset{T}{\rightrightarrows} M_1[Y^*_{M_1}]$, then $M_2 \overset{T^*}{\rightrightarrows} M_1[Y^{**}_{M_1}]$.

(2) Suppose $M_2 \overset{T}{\rightrightarrows} M_1[Y^*_{M_1}]$ and $M_3 \overset{T}{\rightrightarrows} M_2[\overline{Y}_{M_2}]$. Let $Y^*_{M_2}$ be the subset of Y_{M_2} consisting of all memory states that are the M_1-associates of at least one member of $Y^*_{M_1}$. If $\tilde{Y}_{M_2} = Y^*_{M_2} \cap \overline{Y}_{M_2}$ and $\tilde{T} = T \cap \overline{T}$ are nonempty, then $M_3 \overset{\tilde{T}}{\rightrightarrows} M_1[\tilde{Y}_{M_1}]$, where $\tilde{Y}_{M_1} \subseteq Y^*_{M_1}$ consists of those states having at least one M_1-associate in $\tilde{Y}_{M_2}$.

2.3.4. Reduction of an Abstract Scheme to Form Γ. To reduce a scheme $M^{n,s}(\tau,X)$ into a form more nearly approximating the program schemes of Lyapunov, a few more conditions are imposed. Attention is focused on the values taken by an element $B^k(\tau,X)$ of a scheme $M^{n,s}(\tau,X)$ until a concluding triple is reached; that is, if $X \in Y_M$, let $T_k(M,X_0) = \{B^k(\nu,X_0)\}$ for $\nu = 0, \ldots, \nu_M(X_0)$. The element $B^k(\tau,X)$ is called an operator over the set $Y^*_m \subseteq Y_M$ if $M_k(M,Y^*_M) = \bigcup_{X_0 \in Y^*_M} T_k(X,X_0)$ is a subset of $\mathfrak{A}$. The notions of logical condition and a functor over Y^*_M are similarly defined. A logical

element $B^k(\tau,M) \in M^{n,s}(\tau,X)$ is called directed if the logical conditions of the set $T_k(M,Y_M^*)$ differ from one another only in their predicates. $B^k(\tau,X)$ is called M-correct if one of the following conditions hold:

(a) $B^k(0,X) \notin W$,

(b) $B^k(0,X) = (\Phi^k, \iota, m) \in W$, $0 \leq \iota, m \leq n + s$.

A scheme $M^{n,s}(\tau,X)$ is said to be representable over $Y_M^* \subseteq Y_M$ if each element is either an operator, or a directed logical element, or an M-correct functor.

The significance of the representable schemes lies in this: They are easily and naturally put into a form recognizably similar to that of program schemes, the form (Γ), to use Podlovchenko's term. Consider, then, a scheme $M^{n,s}(\tau,X)$ that is representable over $Y_M^* \subseteq Y_M$. With each logical element $B^i(\tau,X) = p^i(\tau,X), \iota^i)$ over Y_M^* there is associated a natural number r_i and two arrows indexed with r_i: an entering arrow $\downarrow^{r_i}$ and an exiting arrow $\uparrow^{r_i}$. It is understood that $r_i = r_j$ if and only if $i = j$. The element $(p^i(\tau,X),\iota^i)$ becomes $p^i(\tau,X)\uparrow^{r_i}$, and $\downarrow^{r_i}$ is placed immediately before (to the left of) the element $B^{\iota^i}(\tau,X)$ if $0 \leq \iota^i \leq n$, and to the right of the element $B^n(\tau,X)$ if $n < \iota^i$. In place of each functor over Y_M^*, that is, an element of the form $B^k(\tau,X) = (\Phi^k, \iota^k, m^k)$, $0 \leq k \leq n + s$, one writes the expression Φ^k and places the number k beneath the element $B^{\iota^k}(\tau,X)$ and above the element $B^{m_k}(\tau,X)$. A vertical stroke is placed just before the element $B^{n+1}(\tau,X)$,

and all occurrences of commas in the scheme are omitted. The result of these alterations is the form (Γ) of the scheme $M^{n,s}(\tau,X)$ over Y^*_M. As Y^*_M changes, the form (Γ) of $M^{n,s}(\tau,X)$ may change. Also, it is obvious that there are several forms (Γ) of a scheme $M^{N,s}(\tau,X)$ even over a single set Y^*_M; these several forms result from notational changes such as choosing different indices for the logical conditions, etc.

Let us take the previous example $M^{6,0}$ of a scheme and rewrite it in a form (Γ). Application of the above rules results in

$$M^{6,0} = A \overset{3}{\downarrow} \overset{2}{\downarrow} Bp_1 \overset{1}{\uparrow} Cp_1 \overset{2}{\uparrow} D \overset{1}{\downarrow} p_2 \overset{3}{\uparrow}.$$

Next, suppose the sequence

$$M^{4,1}(0,X) = (\Phi^0,5,1),\ A,D,(\Phi^3,1,),\ (p_1,1),\ B$$

defines a scheme $M^{4,1}(\tau,X)$, where A, D, B $\in \mathcal{A}$, Φ^0, $\Phi^3 \in \mathcal{F}$, p $\in \mathcal{B}$. Here the operators D and B are assumed constant. Assuming that scheme $M^{4,1}(\tau,X)$ is representable for some $X_0 \in Y_{M^{4,1}}$, an instance of the form (Γ) is

$$M^{4,1}(\tau,X) = \Phi^0 \overset{1}{\downarrow} A\left(\begin{matrix}0,3\\ \tau\\ 3\end{matrix}, X\right) D\Phi^3 p \overset{1}{\uparrow} \Big| \begin{matrix}B.\\ 0\end{matrix}$$

2.3.5. Transformations and Parametrization of Schemes. It is to be observed that the starting form $M^{n,s}(0,X)$ of a sequence $M^{n,s}(\tau,x)$ completely defines the operation and structure of this sequence. Consequently, to discuss

transformations of such sequences, it suffices to discuss transformations of their starting forms.

A segment $\varphi_{k,r} = B^k(0,X), \ldots, B^{k+r}(0,X)$, $k + r \leq n$ of given $M_{n,s}(0,X)$ is said to be free, roughly speaking, if among the first $n + 1$ elements of the sequence $M^{n,s}(0,X)$ which are functors or logical conditions not in the segment, the affect of these elements falls outside the segment $\varphi_{k,r}$. More precisely, $\varphi_{k,r}$ is said to be <u>free</u> if

(a) for each functor $B^i(0,X) = (\Phi^i, \ell^i, m^i) \in M^{n,s}(0,X)$, $0 \leq i \leq n$, it is true that ℓ^i, $m^i \neq k, k+1, \ldots, k + r$; and if

(b) for each logical condition $B^i(0,X) = (p^i, \ell^i)$ in $M^{n,s}(0,X)$ that does not occur in the segment, it is true that $\ell^i \neq k, k + 1, \ldots, k + r$.

Consider a sequence $M^{n,s}(0,X)$ capable of being represented in the form (Γ), a free segment $\varphi_{k,r}$ of the same, and a sequence $\Psi_{\tilde{r}} = \tilde{B}^0, \ldots, \tilde{B}^{\tilde{r}}$, $\tilde{r} \geq -1$, of elements of a $\mathcal{A} \cup V_{\mathcal{B}} \cup W_{\mathcal{H}}$. Suppose $M^{n,s}(0,X)$ is put into the form (Γ), and the segment $\varphi_{k,r}$ is replaced by the sequence $\Psi_{\tilde{r}}$, giving a sequence $\tilde{M}^{n+\tilde{r}-r,s}(0,X)$, which defines a sequence $\tilde{M}^{n+\tilde{r}-r,s}(\tau,X)$. In brief,

$$\tilde{M}^{n+\tilde{r}-r,s}(\tau,X) \equiv M^{n,s}(\tau,X)\ \{\Psi_{\tilde{r}} \rightarrow \varphi_{k,r}\}.$$

Consider now the alteration of a memory state at a single cell $z_0 \in Z$; that is, for $X_0 \in Y$ and $x \in G$, consider the memory state X such that $X(z) = X_0(z)$ for

$z \neq z_0$ and $X(z_0) = x$; write $X_0\,[x \to z_0]$.

If M is a scheme, the cell $z_0 \in Z$ is said to be M-free over $Y_M^* \subseteq Y_m$ if for $X_0 \in Y_M^*$ and $x \in G$, $X_0[x \to z_0] \in Y_M$ and

(a) $L_M(X_0) = L_M(X_0[x \to z_0])$;

(b) for each $0 \leq \tau \leq v_M(X_0) + 1$, the memory-states $X(\tau,X_0)$ and $X(\tau,X_0[x \to z_0])$ differ at most on z_0, and all $X(\tau,X_0)$ and all $X(\tau,X_0[x \to z_0])$ have the same value on z_0. The set $Z(M,Y_M^*)$ of all cells in Z that are not M-free over Y_M^* are said to be M-used over Y_M^*. Now, we associate with each scheme M_0 and set $Y_{M_0}^* - Y_{M_0}$ the set $\mathcal{M}(M_0,Y_{M_0}^*)$ of all schemes M such that $M \supseteq M_0[Y_{M_0}^*]$, where $\mathcal{J}$ is M_0-used over $Y_{M_0}^*$. If $M \in \mathcal{M}(M_0,Y_{M_0}^*)$ and some rule maps M into a scheme M^*, this rule is called a <u>transformation</u> of M into M^*, and is said to be M_0-invariant over $Y_{M_0}^*$ if $M^* \in \mathcal{M}(M_0,Y_{M_0}^*)$.

For any integral $m \geq 1$, consider a finite set $I_m = \{(i_1,i_2,\ldots,i_m)\}$, where the i_s assume only integral values. Let $I_m^{(s)}$ denote the projection of the set I_m out the i_s-axis. For $m > 1$, if in each set $I_m^{(s)}$, $s = 1,2,\ldots,m$, there is a point that is the image of at least two points of I_m, I_m is said to be <u>nonuniform</u>. By definition, I_1 is non-uniform. A parametrization of a set K is a mapping q of an I_m onto K, and i_1, i_2, ..., i_m are said to be parameters of the set K. If q is one-to-one, the parametrization is called <u>restricted</u>. A region $J_s = \{(j_1,j_2,\ldots,j_s)\}$ is an

extension of the region $\{(i_1,i_2,\ldots,i_m)\}$ if $s \geq m$, and for each point $(i_1,i_2,\ldots,i_m) \in I_m$ there is a point $(j_1,j_2,\ldots,j_s) \in J_s$ such that $i_1 = j_1, i_2 = j_2, \ldots, i_m = j_m$.

The introduction of the notion of a parametrization enables the elements of a scheme to be represented in more customary fashion. Consider a scheme $M^{n,s}(\tau,X)$. Let $T^*_\ell(M,X_0) = \{B^\ell(\tau^\ell,X_0) | \tau^\ell \in \nu_\ell(X_0),\ X_0 \in Y_M,\ 0 \leq \ell \leq n+s\}$. Suppose that $T^*_\ell(M,X_0)$ is parametrized by use of the region $J_u = \{(j_1,j_2,\ldots,j_u)\}$. For $\tau^\ell \in \nu_\ell(X_0)$, each pair (τ^ℓ,X_0) defines an element $B^\ell(\tau^\ell,X_0) \in T^*_\ell(M,X_0)$. Let $(j_1'^0,\ldots,j_u'^0)$ be a preimage of $B^\ell(\tau^\ell,X_0)$ in J_u. Define a correspondence Π by

$$\Pi(\tau^\ell,X_0) = (j_1^0,j_2^0,\ldots,j_u^0),\ \tau^\ell \in \nu_\ell(X_0).$$

This done, the element $B^\ell(\tau^\ell,X_0)$ can be represented as $B^\ell_{j_1j_2\cdots j_u}$ and a logical element as $(p^\ell_{j_1j_2\cdots j_u},\lambda)$.

2.3.6. Reduction of an Abstract Scheme to a Program Scheme. A scheme M_0 such that its starting form consists only of elements from a $V_{\mathcal{B}}$ is called a calculation scheme. Any member of the set $\mathcal{M}(M_0,X_0)$ is a program scheme corresponding to the calculation scheme M_0 and admissible state X_0.

To take a particular case, suppose $\overline{A} \in \mathcal{A}$ and

$$M_0^{n,0} = \underbrace{\overline{A}\ .\ \overline{A}\ \ldots\ \overline{A}}_{n+1 \text{ times}}\ .$$

It is easy to construct a program scheme corresponding to $M_0^{n,0}$ in which the operator $\bar{A}$ occurs just once. Let $\delta^{(1)}$ be a function of n_0 variables ($n_0 \geq 1$) that maps G^{n_0} into G. The function χ defined by

$$\chi(\delta^{(1)};\ z_1,\ z_2,\ \ldots,\ z_{n_0};\ X(z)) = \delta^{(1)}(X(z_1),\ X(z_2),\ \ldots,\ X(z_{n_0}))$$

takes Y into G. Let $\tilde{z}$ be a member of Z. The function

$$X^*(z) = \begin{cases} X^{(z)} \text{ if } z \neq \tilde{z}, \\ \chi(\delta^{(1)};\ z_1,\ \ldots,\ z_{n_0};\ X(z)) \text{ if } z = \tilde{z}, \end{cases}$$

$X \in Y$, defines an operator, called an <u>elementary operator</u>, and is denoted by

$$A(\delta^{(1)};\ z_1,\ z_2,\ \ldots,\ z_{n_0},\ \tilde{z},\ X).$$

Taking n = 3,

$$M^{3,0} = (z' \rightarrow z_0)\overset{1}{\downarrow}\ \bar{A}\ A(\delta_1^{(1)};\ z_0,\ z',\ z_0;\ X)\ p(z_0 = z'')\overset{1}{\downarrow}.$$

In this case, $n_0 = 2$, $\tilde{z} = z_0$, and $\delta_1^{(1)}(x_1,\ x_2) = x_1 + x_2$. Now, given an initial state X' such that

$$X'(z') = 1,\ X'(z'') = n + 2,$$

the operator $\bar{A}$ is executed n + 1 times. It follows from the

definition of M_0—used that the condition

$$z_0,\ z',\ z'' \notin Z(M_0,X_0),$$

where X_0 is a state, reduces to $M^{3,0} \in \mathcal{M}(M_0,X_0)$. Also, a memory—state X' which besides having the designated values on z' and z'' also agrees with X_0 for $z \in Z(M_0, X_0)$, is an M_0-associate of X_0. If the program scheme is to be considered only with respect to such states, it assumes the form

$$M^{3,0} = (1 \rightarrow z_0) \overset{1}{\downarrow} \overline{A}\ f(z_0)p(z_0 = n + 2),$$

where $f(\tilde{z})$ is short for the operator $A(\delta_1^{(1)};\ \tilde{z},\ z,\ \tilde{z};\ X)$ with $X(z) = 1$ for $z \in Z$.

Thus, the program scheme $M^{3,0}$ consists of just four elements, no matter how large n is. Notice, however, that if $\nu_{M_0}(X_0) = n$, then $\nu_{M^{3,0}}(X') = Bn$.

2.4. Algebraic Treatment of Operator Programming

A. A. Lyapunov, who more than anyone else should be considered the originator of the operator approach to programming, feels that isolation of the notions of operator programming from the body of mathematics condemns the former to sterility. The problem, as he sees it, is to define the mathematical principles basic to programming so that the methods of the more suitable branches of

mathematics can be brought to bear. He judges that the work of Yanov, Podlovchenko, and others on the transformations of program schemes reveals a similarity between these transformations and the general classes of transformations studied in the theory of categories [22]. This paper is brief and sketchy, with a good number of the proofs omitted. It nevertheless occupies a position relative to the algebraic papers analgous to that of Lyapunov's [21] in the whole series: it gets the show on the road. It sets the stage for the topics pursued in ensuing papers, especially those of Glebov. In the sequel, the order of topics taken up in Lyapunov's paper has been somewhat altered to provide an easy passage into Glebov's [14].

2.4.1. Basic Notions. If Ω, G, and M designate the memory, the set of states of the memory cells, and the set of memory-states, respectively, all as defined in Podlovchenko's [29], a mapping $A:M_1 \to M_2$, $M_1 \subset M$, $M_2 \subset M$ is called an operator over Ω. The composition of operators is defined as above; that is, if $A:M_1 \to M_2$ and $B:M_2 \to M_3$, then $A \cdot B:M_1 \to M_3$ with, by definition, $AB(f) = B(A(f))$, $f \in M_1$. The class of operators over Ω constitutes a category K_Ω. The objects of K_Ω are the subsets M' of M; the abstract maps are the operators over Ω, and the identity associated with each object M' is the identity operator defined on M'.

Given a set $U = \{u\}$ and $M_1 \subset M$, a mapping $p:M_1 \to u$ is called a predicate over Ω. Suppose a system A^u of operators

is given that are indexed by the members u of some set $U_1 \subset U$. It is allowed that $A^{u_1} = A^{u_2}$ even though $u_1 \neq u_2$. The operator B defined by p—composition is prescribed by the following equation:

$$B(f) = A^{u}(f) \text{ where } p(f) = u.$$

If $P = \{p\}$ is a class of predicates over Ω, and K is a class of operators over Ω, K is said to be closed with respect to P if K is closed with respect to p—composition of elements for $p \in P$.

Lyapunov employs this closure definition to introduce an equivalence relation between classes of predicates over Ω which enables him to characterize these classes algebraically. Thus, class P_1 of predicates over Ω is said to be subordinate to class P_2 if each class of operators that is closed with respect to P_1 is closed with respect to P_2. The relation R defined by $P_1 R P_2$ if and only if P_1 is subordinate to P_2 and P_2 is subordinate to P_1 is an equivalence relation. Obviously, the relation of P_1 being subordinate to P_2 is a partial order relation and induces a partial ordering on the set of equivalence classes of predicates over Ω.

The smallest algebra of subsets of M that includes all preimages of elements u in the range of p for all $p \in P$, is termed the algebra generated by P. Notation: $\mathcal{A}_P$.

The fact is that class P_1 is subordinate to class P_2

if and only if $\mathcal{A}_{P_1} \subset \mathcal{A}_{P_2}$, and that the classes of equivalent predicates over Ω constitute a complete lattice under the aforementioned partial ordering. Suppose besides K and P we are given a class H = {h} of functions mapping U into the natural numbers and an expression STOP. A K–term is an ordered pair (A,m) where A ϵ K and m is a natural number. A P–term is an ordered pair (p,h), p ϵ P and h ϵ H. STOP is a 0–term. It is assumed that all K–terms, P–terms, and the 0–term are distinct. By a term is meant a K–term or a P–term, or the 0–term.

A logical scheme is a sequence of terms $\alpha_1, \alpha_2, \ldots, \alpha_k$, and defines an operator B over Ω according to the following prescription. Let f_0 be the starting memory state.

(a) Initially, the term with smallest subscript is marked.

Suppose the term α_i has been marked and the current memory state is f.

(b) If α_i is a K–term, α_i = (A,m), then the memory state f is changed to A(f) and the next term marked is α_m. If A(f) is undefined, then the process terminates and $B(f_0)$ is undefined.

(c) If α_i is a P–term, α_i = (p,h), then the memory state continues to be f and the next term marked is $\alpha_{h(u)}$, where p(f) = u.

(d) If α_i is the 0–term and f is the current memory state, then the process terminates and $B(f_0)$ = f.

(e) If by application of the above, the process of marking next terms continues indefinitely, but from some point on the current memory state remains f, then $B(f_0) = f$. If the current memory state never becomes constant, then $B(f_0)$ is undefined.

Logical schemes admit of an algebraic characterization when Ω and G are finite. Suppose this is the case, and that K contains the identity operator on Ω. Lyapunov states that the set of operators representable by logical schemes is the smallest category (in the obvious sense) of operators that includes K and is closed with respect to p—composition for $p \in P$.

Since a logical scheme defines an operator over Ω, and hence alters the state of the memory, one may think of an auxiliary memory $\Omega' = \{c'\}$ in which logical schemes are stored as exercising a measure of control over Ω, briefly as a control memory. Let $G' = \{g'\}$ and $M' = \{m'\}$ be the set of states and memory—states, respectively, associated with Ω'. Associated with each cell c' is the value $\delta(c') = 1$. Assume now a 1 to 1 correspondence m has been established between all terms α and the states g', and that the memory cells have been indexed via a mapping into the natural numbers.

Let $\mathcal{A}(\alpha_1, \alpha_2, \ldots, \alpha_n, \ldots)$ be a logical scheme and N a set of natural numbers serving as an index set for the terms of $\mathcal{A}$. Let $\Omega'_N \subset \Omega'$ be the set of cells with

index set N. By a code of the logical scheme is meant the memory state $m_{\mathcal{A}}$ such that

$$m_{\mathcal{A}}(c) = \begin{cases} m(\alpha_n) & \text{if } c = c_n, \\ \text{STOP} & \text{if } c \notin \Omega'_N. \end{cases}$$

Thus the logical scheme $\mathcal{A}$ is encoded in the auxiliary memory Ω'. The changes effected in the state of Ω by operation of $\mathcal{A}$ now take the following appearance in terms of the code:

(a) Initially, $\delta(c_1) = 1$.

(b) The term is executed whose encoded form occupies cell c such that $\delta(c) = 1$. According to the process defined earlier, the next term to be executed is defined. Let c' be the cell containing the code for this next term. Then $\delta(c') = 1$.

(c) If the code for the 0-term appears in cell c such that $\delta(c) = 1$, or if the memory-state f of Ω becomes constant from some point on, the process of transforming Ω terminates and $B(f_0) = f$, where f_0 is the initial memory-state.

This relation of an auxiliary memory Ω' controlling a memory Ω can obviously be iterated. Accordingly, Ω' is said to be of rank 1 over Ω, and if a memory Ω^n of rank n over Ω has been defined, then a memory Ω^{n+1} controlling Ω is said to be of rank n + 1 over Ω.

Lyapunov also undertakes to characterize algebraically classes of operators that specialize to such things as readdress operators. To this end, consider a set $Z = \{z\}$

and a set $\Delta = \{\delta\}$ of partial transformations of Z. $Z_\delta = \{z | z \in Z$ and $\delta(z)$ is defined$\}$. By the sum of transformations δ_1 and δ_2 is meant the composition of δ_1 with δ_2; that is, $(\delta_1 + \delta_2)(z) = \delta_2(\delta_1(z))$. Denoting $\delta(z)$ by $z \dot{+} \delta$ we have $z \dot{+} (\delta_1 + \delta_2) = (z \dot{+} \delta_1) + \delta_2$ whenever $z \dot{+} \delta_1 \in Z_{\delta_2}$. In the sequel it is assumed that Δ is closed with respect to composition and that the latter is commutative whenever both $\delta_1 + \delta_2$ and $\delta_2 + \delta_1$ are defined. Note that composition is associative if either $\delta_1 + (\delta_2 + \delta_3)$ or $(\delta_1 + \delta_2) + \delta_3$ is defined.

Operators over Ω are, by definition, operators of rank 1. A mapping of subsets of categories of operators of rank n into themselves are operators of rank n + 1. These mappings also form a category. Now consider a mapping $n: z \to K_\Omega$. Writing $n(z) = A_z$ and assuming $\delta \in A$ and $z \dot{+} \delta \in Z$, we define

$$B_\delta(A_z(f)) = A_z \dot{+} \delta(f).$$

Note that B_δ is of rank 2 and is called a δ-shift of M. Note also that B_{δ_1} and B_{δ_2} commute.

Next, consider a function Ψ defined on $M_1 \subset M$, and define Ψ to be Ω_1-consistent if

(a) $f_1 \in M_1$ and $f_1(x) = f_2(x)$ for all $x \in \Omega_1$ implies that $f_2 \in M_1$,

and

(b) f_1 and $f_2 \in M_1$ and $f_1(x) = f_2(x)$ for all $x \in \Omega_1$, then $\Psi(f_1) = \Psi(f_2)$.

By an Ω_1-restriction of an Ω_1-consistent function Ψ defined on $M_1 \subset M$ is meant a function $\overline{\Psi}$ whose domain includes M_1 such that $\overline{\Psi}(\overline{f}) = \Psi(f)$ if $\overline{f}(x) = f(x)$ for $x \in \Omega_1$ and $f \in M_1$. Notation: $\overline{\Psi}(\overline{f}) = \Psi(f/\Omega_1)$.

Lyapunov lists the following properties of Ω_1 restrictions of Ω-consistent predicates or operators over Ω:

(a) If some class of Ω_1-consistent predicates over Ω is invariant with respect to some operation over the values of the predicates, its Ω_1-restriction has the same property.

(b) If the set of Ω_1-consistent operators form a category, then their Ω_1-restrictions also form a category, the Ω_1-restriction of the original category.

(c) A mapping taking each operator of some category to its Ω_1-restriction maps the given category onto its Ω_1-restriction.

(d) An Ω_1-restriction of a class of Ω_1-consistent operators that is closed with respect to the class P of Ω_1-consistent predicates may be represented as a class of operators closed with respect to the Ω_1-restriction of class P.

2.4.2. (n, m)-Operators and Predicates. Suppose a mapping $\nu: Z \to \Omega$ is given. An n-tuple of cells $(x_1, x_2, \ldots, x_n)$ and memory state f and a function F defined on $G^n = \underbrace{GxGx\ldots xG}_{n \text{ factors}}$ defines an n-function of special form of the memory state if

$$\varphi(x_1,\ldots,x_n|f) = F(f(x_1),\ f(x_2),\ldots,f(x_n)).$$

If F in the above takes G^n into some set U, the equation defines a predicate of special form. An n—function of special form is called an (n,m)—operation if $F:G^n \to G^m$.

A special case of an n—function that is of interest is an (n,m)—operator. Suppose $y_1,\ldots,y_m$ are m distinct cells and an (n,m)—operation $\varphi(x_1, x_2, \ldots, x_n|f)$ is given. Then the equation

$$A(f) = f_1,$$

where $f_1(x) = g_i$ if $x = y_i$, and $f_1(x) = f(x)$ if $x \neq y_i$ for $i = 1, 2, \ldots, m$, defines an (n,m)—operator of special form.

Notation: $A(\varphi(x_1,x_2,\ldots,x_n|f), y_1,y_2,\ldots,y_m) = f_1$.

Let $x_i = x_z^i$ for $z \in Z$, $\delta \in \Delta$, $z \dot{+} \delta \in Z$. The operators

$$B_{i,\delta}(\varphi(x_1,\ldots,x_{i-1},x_z^i,x_{i+1},\ldots,x_n|f))$$

$$= \varphi(x_1,\ldots,x_{i-1},\ x_{z+\delta}^i,x_{i+1},\ldots,x_n|f),\ i \leq n,$$

$$B_{n+i,\delta}(A(\varphi(x_1,x_2,\ldots,x_n|f),y_1,\ldots,y_{i-1},y_z^i,y_{i+1},\ldots,y_m))$$

$$= A(\varphi(x_1,\ldots,x_n|f),y_1,\ldots,y_{i-1},y_{z+\delta}^i,y_{i+1},\ldots,y_m)),\ i \leq m$$

are called elementary readdress operators.

A readdress operator is a product of elementary readdress operators. It is clear that composition of readdress operators is commutative. The reader will notice that readdressing is a special case of the shift operation, while operators and predicates of special form are special cases of Ω_1—restrictions.

2.4.3. Synthesis of (n,m)—Operators. In the case of (n,m)—operators, one has a quantitative measure of the complexity of these operators, namely, the numbers m and n. This invites an attempt to represent members of this class in terms of the more simple members. The problem is that of the synthesis of an operator from given, simple operators by representation of the former as a product of the latter; this is considered by Glebov in [14]. Since the problem is one of synthesis, it is not surprising that the proofs of the theorems of the paper are constructive and some genuine computational questions arise concerning the efficiency of the method of synthesis.

The synthesis problems treated by Glebov are as follows:

(1) Synthesis of (n,m)-operators from (n,1)-operators

(2) Synthesis of (n,1)-operators from (m,1)-operators.

(3) Synthesis of (n,m)-operators from (1,1)-operators with 1-predicates.

Relative to (3), an operator is said to be obtained by synthesis of operators and predicates of a given class if it can be obtained from this given class by means of the operations of multiplication and p—composition.

In answer to (1) Glebov proves: If a memory $\Omega = \{x_1, x_2, \ldots, x_n\}$ consists of n cells each of which admits of a finite number of states, then any operator over Ω can be synthesized from (n,1)—operators. In the course of the proof of this assertion, Glebov introduces the notions of a transposition operator and a reduction operator.

A <u>transposition operator</u> $[g(x), g'(x)]$ is a mapping of Ω that maps the element g and g' of the set M of memory states into one another and leaves all other members of M fixed. A <u>reduction operator</u> $[g,g']^*$ is a mapping of M that takes g to g' and leaves all other members of M fixed.

To consider the question of estimating the efficiency of the method of synthesis, a special type of transposition and reduction operator is defined. An elementary transposition operator $(g|g')_i$ is defined only when $g \neq g'$ and g differs from g' in only its value on some one cell x_i; that is, $g(x_i) \neq g'(x_i)$ and $g(x_j) = g'(x_j)$ for $j \neq i$. Under these conditions $(g,g')_i$ interchanges g and g', and elementary reduction operator $(g|g')_i^*$ is defined as was an elementary transposition operator with the difference that, when defined, $(g|g')_i^*$ takes g to g' and leaves all other members of M invariant.

Let N be the least integer such that every operator Ω is representable as a product of elementary transposition and reduction operators. Glebov shows that if each of the n memory cells admits of q states, then

$$C(n,q)q^n \leq N \text{ where } 3/7 < C(n,q) < 1.$$

He expresses his belief that the upper bound is very crude.

To handle the general case of (1) he proves: If Ω is an arbitrary memory each cell of which admits of a finite number of states, then any (n,m)—operator over Ω can be synthesized from (n,1)—operators over Ω. Relative to consideration (2) is the fact that under the stated conditions on Ω, there exist (n,1)—operators over Ω that cannot be synthesized from (m,1)—operators over Ω when $m < n$. Also, he establishes that if Ω consists of n cells, any operator over Ω can be synthesized from (1,1)—operators and 1—predicates.

Glebov also considers the question of synthesizing operators over a memory Ω from (m,1)—operators by use of additional memory cells. Let $\Omega = \{x_1, x_2, \ldots, x_n\}$ be a given memory, the cells of which admit of a finite number of states constituting the set G. $\Omega' = \{x_{n+1}, x_{n+2}, \ldots, x_{n+k}\}$ and $G' = \{0,1,\ldots,s-1\}$ denote the working memory and states, respectively, and $M(\Omega)$ and $M(\Omega')$ represent the sets of memory—states of Ω and Ω', respectively. Take $\overline{G} = G \cup G'$,

$\overline{\Omega} = \Omega \cup \Omega'$. Then $M(\overline{\Omega}) = \{\overline{g} | \overline{g}:\Omega \to G, \overline{g}:\Omega' \to G'\}$. Define $M^0(\overline{\Omega}) = \{\overline{g} | \overline{g} \in M(\overline{\Omega}), \overline{g}(x_i) = 0 \text{ for } x_i \in \Omega'\}$ and the mapping

$$\sigma:M(\Omega) \to M^0(\overline{\Omega})$$

by $\sigma(g) = \overline{g}$ if $g(x_i) = \overline{g}(x_i)$ for $x_i \in \Omega$, $g \in M(g)$, and $\overline{g} \in M^0(\Omega)$.

σ is a 1–1 mapping of $M(\Omega)$ onto $M^0(\Omega)$ and induces an isomorphism σ' of the category K_Ω of operators over Ω into some category K^0 of operators over $\overline{\Omega}$, as follows. The domain and range of $A^0 = \sigma'A$ correspond to the domain and range of A under σ, and A^0 is such that $A(g) \overset{\sigma}{\leftrightarrow} A^0(\overline{g})$ if $g \overset{\sigma}{\leftrightarrow} \overline{g}$.

The problem now takes the form: What are the least values of k and s for which any operator $A^0 \in K^0$ can be synthesized from an (m,1)–operator ($2 \leq m < n$) over $\overline{\Omega}$? A partial answer: At least one working cell is necessary for synthesis of an operator $A^0 \in K^0$ from (2,1)-operators over $\overline{\Omega}$ while a single working cell (k = 1) that admits of two states (s = 2) is sufficient for this purpose.

2.4.4. An Algebraic Equivalent of Synthesis. A method of synthesizing an operator over a memory from relatively simple operators is of potential value in that it is to be expected that programming the simple operators is itself relatively straightforward, and hence the method presumably provides the means for programming the more complex operator. In his interesting paper [15] Glebov develops

an algebraic equivalent of this situation and gives some applications. The problem takes the following form: given an operatory category K and a set $E \subset K$, characterize those members A of K that are representable as products of members of E. This brings us to a brief mention of the subcategories of a category.

A subcategory K_1 of a category K is said to be proper if every identity element of K_1 is also an identity element of K. It is clear that the interaction of an arbitrary set of proper subcategories of a category is a proper subcategory.

In treating the stated problem, we may assume that E contains no identity elements. By the proper subcategory K(E) generated by a set E is meant the minimal proper subcategory containing E.

It is important to know when the sets of operators that can be synthesized from two given basic sets are the same. This consideration leads to the following definition: sets E_1 and E_2 are said to be equivalent if they generate the same minimal subcategory, that is, if $K(E_1) = K(E_2)$. To investigate the conditions under which this equation holds, some auxiliary notions are needed.

Suppose some operator category K_α is given, consisting of operators $A:M_1 \to M_2$, where M_1 and M_2 are subsets of some set M_α. We define a subordination relation $\overset{r}{<}$ on $M_\alpha \times M_\alpha$ by these two axioms:

(1) If $f_1 \overset{r}{<} f_2$ and $f_2 \overset{r}{<} f_3$, then $f_1 \overset{r}{<} f_3$.

(2) If $f_1 \overset{r}{<} f_2$, then $A(f_1) \overset{r}{<} A(f_2)$ for any $A \in K_\alpha$ defined on f_1 and f_2. An example of such a subordination relation is $f_1 \overset{1}{<} f_2$ if and only if $f_1 = f_2$. Another example is $f_1 \overset{2}{<} f_2$ for all $(f_1, f_2) \in M_\alpha \times M_\alpha$.

Let $\mathcal{M} \subset K_\alpha$. By a quasi-invariant of $\mathcal{M}$ is meant any $f \in M_\alpha$ such that $Af \overset{r}{<} f$ for all $A \in \mathcal{M}$. $S(\mathcal{M})$ designates the set of all quasi-invariants of $\mathcal{M}$ and hence

$$S(\mathcal{M}) = \bigcap_{A \in \mathcal{M}} S(A),$$

where $S(A)$ is the set of all quasi-invariants of the operator A.

If σ is a homomorphism of a category K into K_α, $E \subset K$, and $R = (K_\alpha, \overset{r}{<}, \sigma)$, then the set $R(E) = S(\sigma E)$ is called the R-characteristic of E, and the members of $R(E)$ are called σ-quasi-invariants. These preliminaries put one in a position to define sets E_1 and E_2 (each having no identity elements) as being R-equivalent if $R(E_1) = R(E_2)$. Glebov proves that $E_1, E_2 \subset K$ are equivalent if and only if they are R-equivalent.

The R-characteristic of a set E is a function of the operator category K_α, the subordination relation $\overset{r}{<}$, and the homomorphism σ. Thus for each triple $R = (K_\alpha, \overset{r}{<}, \sigma)$ we have a possibly different R-equivalence criterion for the subsets of a category K. In this connection Glebov defines

an R_1—criterion to be weaker than an R_2—criterion, $R_1 \leq R_2$ if any pair (E_1, E_2) of subsets of K that are R_2—equivalent are also R_1—equivalent. (One would think that Glebov should say, "no stronger than," but he actually does say, "weaker than"). As the reader doubtlessly anticipates, an R_1—criterion and an R_2—criterion are equipotent if $R_1 \leq R_2$ and $R_2 \leq R_1$. R(K) denotes the set whose elements are classes of equipotent R—equivalence criteria.

Concerning the structure of R(K), Glebov proves the following theorem: The set R(K) is a complete lattice.

2.4.5. An Example. Glebov next turns to an application of these ideas. Consider a memory Ω consisting of just three cells, x_1, x_2, x_3, each of which admits of being in state 0 or state 1. The set $M = \{f\}$ of memory—states thus consists of eight elements. Consider the operator category K of all invertible operators $A:M \to M$ and the set $E \subset K$ of all (2,1)—operators. In this example, K is the symmetric group of order 8, while the subcategory K(E) is the subgroup of order $8!/30 = 1344$.

The problem is to find necessary and sufficient conditions for $A \in K$ to be a member of K(E). A theorem of Glebov [14] states that any (n,m)—operator over a memory Ω, the cells of which admit of a finite number of states, can be synthesized from (n,1)—operators over Ω. Inspection of the proof reveals that if the given operator is invertible, the factors can be taken to be invertible.

Consequently K(E) contains all invertible (2,2)—operators, and the problem reduces to finding necessary and sufficient conditions for arbitrary $A \in K$ to be representable in terms of invertible (2,2)—operators. This amounts to determining an R_1-equivalence criterion for category K such that every subcategory K_1 of K that is not contained in K(E) is not R_1-equivalent to K(E).

To this end we represent the set M by the set $M' = \{0, 1, \ldots, 7\}$, associating with state g the number $n = \sum_{i=1}^{3} g(x_i)2^{i-1}$, and the members of K can hence be given the following cyclic form:

$$A_1 = (04)(15)(26)(37),\ A_2 = (145)(367),\ A_3 = (246)(357),$$
$$A_4 = (05)(27),\ A_5 = (04)(26),\ A_6 = (01)(45).$$

Glebov divides the invertible (2,2)—operators into three types, characterized by the matrices

$$T_1 = \begin{pmatrix} 0123 \\ 4567 \end{pmatrix},\ T_2 = \begin{pmatrix} 0145 \\ 2367 \end{pmatrix}$$

and

$$T_3 = \begin{pmatrix} 0246 \\ 1357 \end{pmatrix},$$

respectively.

For example, if $A \in K$ and $AT_1 = \begin{pmatrix} A(0)A(1)A(2)A(3) \\ A(4)A(5)A(6)A(7) \end{pmatrix}$,

then A is a (2,2)-operator of type I if and only if AT_1 can be obtained from T_1 by means of column permutations of the above permutations, A_6 is a type I operator, A_1, A_2, and A_4 are type two operators, and A_1 and A_3 are type III.

Let $E = \{e\}$ be the set of all subsets e of M' consisting of four elements, $e = \{n_1, n_2, n_3, n_4\}$. The cardinality of E is $C_4^8 = 70$. Define $M_1 = \{f\}$ to be the set of all subsets f of E consisting of 14 elements. The cardinality of M_1 is C_{14}^{70}. Take K_1 to be the category generated by the set of all invertible operators $A':M_1 \to M_1$ and define for all $f_1, f_2 \in M_1$, $f_1 \overset{1}{<} f_2$ iff $f_1 = f_2$. σ_1 is the homomorphism that takes each $A \in K$ to the operator $\sigma_1 A = A' \in K$ such that A' takes $f = \{(n_1^{(1)}, n_2^{(1)}, \ldots, n_4^{(1)}), \ldots, (n_1^{(14)}, \ldots, n_4^{(14)})\}$ to the elements $f' = \{(An_1^{(1)}, \ldots, An_4^{(1)}), \ldots, (An_1^{(14)}, \ldots, An_4^{(14)})\}$. Set $R_1 = (K_1, \overset{1}{<}, \sigma_1)$.

Consider $f_0 = \{e_1, e_2, \ldots, e_{14}\} \in M_1$, where $e_1 = (0, 1, 2, 3)$, $e_2 = (0, 1, 4, 5)$, $e_3 = (0, 1, 6, 7)$, $e_4 = (0, 2, 4, 6)$, $e_5 = (0, 2, 5, 7)$, $e_6 = (0, 3, 4, 7)$, $e_7 = (0, 3, 5, 6)$, $e_i = e'_{15-i}$ for $i = 8, 9, \ldots, 14$. Let $K(f_0)$ be the set of all $A \in K$ such that f_0 is a $\overset{1}{<}$-quasi-invariant, which, by definition of $\overset{1}{<}$ is the same as requiring $(\sigma_1 A)f_0 = f_0$.

Using these preliminaries, Glebov establishes that $A \in K$ is representable in terms of (2,2)-operators iff f_0 is a σ_1-quasi-invariant of this operator, that is, $K(E) = K(f_0)$.

It is obvious that in general the setting and content of these papers are quite remote from the practices of computer programming. One cannot but share A. A. Lyapunov's opinion that isolation of research in programming from developed branches of mathematics is undesirable and would probably frustrate a fruitful development. His selection of the theory of categories as the framework in which to proceed because of its "close relationship" to the transformations that occur in programming and his seeming expectation of big things to come is perhaps too sanguine. The great generality and skeletal nature of the theory of categories certainly allows room for some description of programming but the value of such a description is another question. In fact, if one looks at the papers it is clear that the depth into which the theory is gone is not very great, and that whenever one gets down to cases (problems), the methods employed don't have much to do with categories but are, of course, combinatorial. It is to be expected that the value of an approach to programming via the theory of categories will remain largely on the descriptive level. It may help to clarify the structure of certain problem areas, but the problems themselves that arise in the development of programming seem more susceptible to the methods of mathematical logic, the theory of algorithms, combinatorics, and graph theory. It will be interesting to see if the work of Glebov will have applications perhaps to automatic programming.

3. OTHER SOVIET EFFORTS

3.1 Preliminary Remarks

Another Soviet effort, related to but largely independent of the theory surveyed in Part 2, has had to do with bringing suitably modified versions of such mathematical disciplines as graph theory and the theory of recursive functions to bear on the problems originating in computer programming. In contradistinction to the theory rooted in the operator method, the analysis is applied not to the classes of operations prescribed by computer programs but to the classes of problems susceptible of programmed solutions. The efforts do not take the transformations recurrent in computer programming as their main point of departure, but concentrate on the very notion of algorithm, seeking to adapt the well known definitions such as Turing machines, normal algorithms, etc., and render them more suitable instruments for studying the problems of programming. The parent paper of this development is L. A. Kaluzhnin's [18]. This paper gives the results of a seminar held in Kiev in 1956 at the Mathematical Institute of the Ukranian S.S.R. Academy of Sciences dealing with the development of computers, and another at Kiev State University, also in 1956, concerned with the theory of algorithms and mathematical

logic. Principal descendants of [18] are Ershov's [10] and [11], and Zaslavskii's [36].

The announced aim of [18] is the development of general methods of preparing mathematical and logical problems for solution with a computer. In constructing a formalism to this end, the stated desirable features are as follows:

(1) Formulation of the general problem as a class of related problems, each of which is described by some word on a (finite) alphabet. The associated class of words is to describe the class of problems, and the words should have some common syntactical property that makes recognition of them easy.

(2) The solution of the general problem is to be a class of words also characterized by certain syntactical properties.

(3) Mechanical rules applicable to initial words and prescribing the transformation of these words into solution words, that is, words composing the solution to the general problem. The collection of these rules may be identified with the algorithm solving the problem.

(4) The algorithm itself is to be expressible as words on some enlargement of the original alphabet.

The entire structure possessing these features is, as is explicitly stated in [7], a language. It is also noted that since computer technology advances constantly, the language should not reflect very closely the structure of any particular machine, lest it soon become out of date. More generally, the following are desirable objectives for the language:

(1) The property of universality: the structure of the language should reflect the basic characteristics of a modern computing machine without being tied to any particular machine.

(2) Standard mathematical and logical algorithms should be readily expressible in the language.

(3) An automatic translator should be provided that translates the language into the machine language of the particular machine being used.

(The reader no doubt recognizes that the features and objectives listed by Kaluzhnin have been realized to a great extent in such higher level languages as FORTRAN and ALGOL.)

One may ask why none of the standard formalisms of the theory of algorithms would serve as an adequate language. In the first place, the theoretically desirable

atomic character of the basic operations of the usual formalisms make the computations of even rather simple functions lengthy and tedious. Again, in complex problems the logical conditions frequently dominate. A satisfactory language ought to make the programming of these logical conditions relatively easy, but the simplicity and economy of the basic operations of Turing machines, systems of equations, and the like, make the simulation of these logical conditions a complicated task. In brief, the very features that make the common formalisms conceptually attractive render them unpractical to be used as models of a programming language. A discussion of Kaluzhnin's [18] follows.

3.2 The Graph Schemes of Kaluzhnin

Let $\mathcal{G}$ be class of finite directed graphs G such that in each G $\epsilon \mathcal{G}$, the following conditions exist:

(1) There is a single distinguished node I, called the input node, of the graph G such that for all other nodes α of G there is at least one directed edge from I to α;

(2) there is exactly one node O from which no arrow (directed edge) begins, the output node of the graph;

(3) there is a partition of all nodes of G other than 0 into two sets:

(a) nodes of the first kind--those from which a single arrow exits,

(b) nodes of the second kind--those from which a pair of arrows exit, one marked with a plus, +, the other with a minus, -.

Suppose that a finite set $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ of operators and a finite set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_m\}$ of discriminators are given. A graph scheme Γ over Φ, Ψ is a graph $G \in \mathcal{G}$ in which each node of the first kind and the output node correspond to some operator φ_i and each node of the second kind corresponds to some discriminator Ψ_j.

3.2.1 Interpretations of Graph Schemes. By an interpretation of an Φ-Ψ-graph scheme is meant a triple $(M, \Phi \to A, \Psi \to F)$, where M is a nonempty set of objects, $\Phi \to A$ is a mapping of the operators Φ onto the class A of mapping A_k of M onto itself, and $\Psi \to F$ is a mapping of the class $\Psi = \{\psi_j\}$ onto the class $F = \{F_j\}$ of properties of members of M. Here

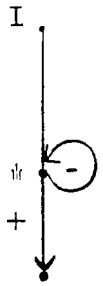
$$F_j^+ = \{m \in M | m \text{ has the property } F_j\},$$

$$F_j^- = \{m \in M | m \text{ doesn't have the property } \overline{F_j}\}.$$

A graph scheme Γ, together with an interpretation $(M, \Phi \to A, \Psi \to F)$, defines an algorithm in the set M as follows: A sequence $m_0 = m, m_1, m_2, \ldots, m_k, \ldots$ is defined, and simultaneously with each m_k there is associated a node of the graph. Accordingly, $m_0 = m$ and is associated with the node I. Suppose $m_0, m_1, \ldots, m_k$, $k > 0$, have been defined and associated, respectively, with $I = n_0, n_1, \ldots, n_k$. If n_k is a node of the first kind and ω_k corresponds to n_k, then $m_{k+1} = A_k(m_k)$, and m_{k+1} is associated with the node n_{k+1} connected with the arrow exiting from n_k. If n_k is a node of the second kind and discriminator ψ_k corresponds to n_k, then $m_{k+1} = m_k$; and if $m_k \in F_k^+$, then n_{k+1} is the node connected with n_k by the exiting plus arrow. If $m_k \in F_k^-$, then n_{k+1} is the node connected with the n_k by the exiting minus arrow. If $n_k = 0$, then the procedure terminates, $m_k = \Gamma(m)$, and m_k is said to be <u>the result of applying Γ to m</u>.

Graph schemes naturally lend themselves to pictorial representations. Here are some very simple examples.

1.

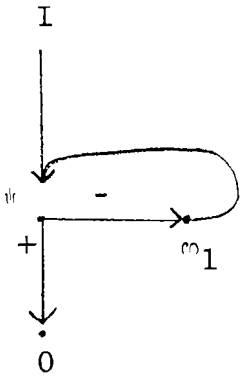


Here $\Phi = \emptyset$ and $\Psi = \{\psi\}$. It is clear that given any interpretation (M, $\Phi \to$ A, $\Psi \to$ F), every m ϵ M is transformed into itself if m has the property F, whereas Γ(m) is undefined if m does not have the property F.

2.

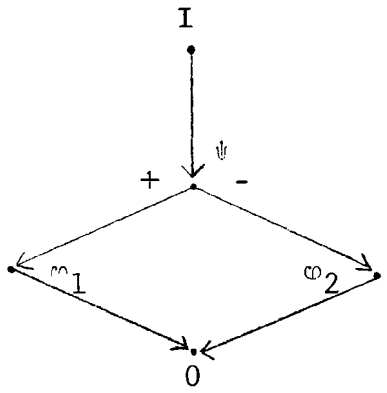
Here $\Psi = \emptyset$, $\Phi = \{\varphi\}$. Apply the operator φ.

3.



Conditional iteration of operator m_1 - m_1 is applied until the current element satisfies property F.

4.



Here $\Psi = \{\psi\}$, $\Phi = \{\varphi_1, \varphi_2\}$. Apply to $m \in M$ the operator φ_1 or φ_2, according as $m \in F_1^+$ or $m \in F_1^-$.

5. An interpretation $(M, \Phi \to A, \Psi \to F)$ is called Markovian if

(1) M is the set of all words on a given finite alphabet.

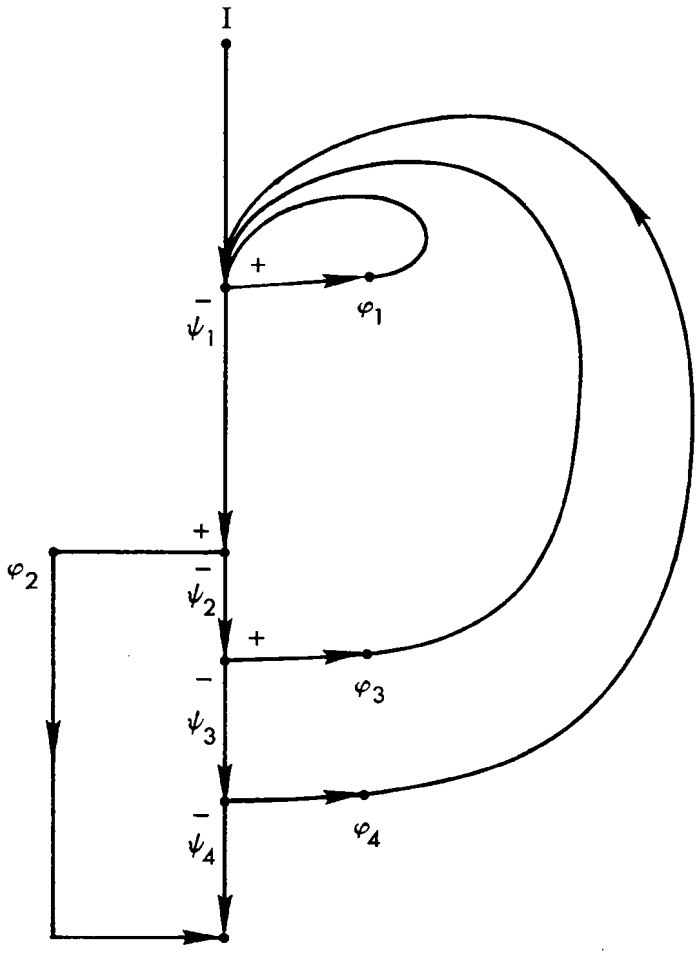
(2) All F_j are properties of the type F_x, where F_x denotes the property that some word contains at least one occurrence of the word x.

(3) All A_i are operations of the type $A_{x \to y}$, where $A_{x \to y}$ stands for the substitution of the word y for the first (left-most) occurrence of the word x in the word A.

Let $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ and $\Psi = \{\psi_1, \ldots, \psi_m\}$ determine a graph scheme. Consider the normal algorithm over some alphabet having the scheme

$$X_1 \to Y_1,$$
$$X_2 \to \cdot\, Y_2,$$
$$X_3 \to Y_3,$$
$$X_4 \to Y_4.$$

Taking $\varphi_1 \to A_{xi} \to Y_1$, $\psi_i \to A_{xi}$ as the members of $\Phi \to A$ and $\Psi \to F$, respectively, the graph-scheme representation of the given normal algorithm is



3.2.2 The Equivalence of Graph Schemes. Kaluzhnin offers two notions of equivalence for graph schemes. The first and obvious definition depends on a given interpretation. Graph schemes Γ_1 and Γ_2 are equivalent with respect to $(M, \Phi \to A, \Psi \to F)$ if for all m in m, $\Gamma_1(m)$ is defined exactly when $\Gamma_2(m)$ is, and in that case $\Gamma_1(m) = \Gamma_2(m)$. The second definition is independent of the interpretation, and reminds one of Yanov's definition of equivalent algorithm schemes. Thus, Γ_1 and Γ_2 are strongly equivalent if for all interpretations the algorithms defined by Γ_1 and Γ_2 are equivalent in the first sense.

Kaluzhnin also mentions the operation of substituting of one graph scheme into another, with a view toward developing an "algorithm algebra." Suppose Γ_1 and Γ_2 are given Φ-Ψ-graph schemes. By the substitution of Γ_2 in Γ_1 is meant the substitution of Γ_2 for each occurrence of an operator φ_i in Γ_1, omitting the input and output nodes of Γ_2. The result of this substitution is itself a Φ-Ψ-graph scheme. In this connection, Kaluzhnin notes the desirability of a substitution operation that could be applied to discriminators as well as operators. To accomplish this, he suggests extending the notion of a graph scheme by providing for two outputs 0^+ and 0^-, arriving at graph schemes of the second kind. However, he observes that if

a graph scheme Γ of the second kind is substituted for an occurrence of a discriminator ψ_i the outputs 0^+ and 0^- of Γ will be not m but $\Gamma(m)$. Thus at ψ_i there occurs not simply a decision but a transformation. As a way out, he suggests the introduction of memory elements into the notion of a graph scheme. Accordingly, in substituting a graph scheme of the second kind for an occurrence of a discriminator, the element m is to be placed in the memory and restored at that one of the outputs 0^+ and 0^- at which it would ordinarily appear. He concludes: "The problem as to the most rational way of introducing such memory elements into the graph-scheme, while preserving the possibility and meaning of the substitution operation, is an important but as yet unsolved question." This consideration forms the basis for the paper of Zaslavskii [36].

3.3 The Operator Algorithms of Ershov

3.3.1 Motivation for the Definition. Ershov's [10] and [11] on operator algorithms represent an attempt to adapt the Markov normal algorithm formalism for rendering the theory more applicable to computer programming, and, in particular, for constructing more adequate programming languages. In [11], a reformulation of the basic concepts of

the theory is carried out, and in [11], applications are discussed. Such a reformulation has been precipitated as a result of the development and use of compilers. Ershov finds the following problems among the more outstanding.

(1) Narrowing the distance between the language in which an algorithm is to be programmed and the rather formal language in which the problem is presented to the compiler. The dissimilarity between the initial description of an algorithm and that of logical schemes has been a particularly important factor in causing the passage from one to the other to be lengthy and time consuming.

(2) Developing a measure of the quality of an algorithm, or of different realizations of some general algorithm, to facilitate optimal programming by selecting the best realization from a class R of algorithms that are equivalent (in some suitable sense). In this connection it is important to "know" the class R as completely as possible by constructing a calculus of transformations generating the members of R.

(3) Developing methods by which the equivalence of algorithms can be effectively recognized. The most general instance of this problem has been shown by H. G. Rice to be recursively unsolvable [30]. The problem thus

becomes that of examining various definitions of "equivalence," hoping to isolate those which make the question of equivalence effectively decidable, and yet are sufficiently broad to include as equivalent a generous portion of those algorithms that would be expected to be equivalent on basic or working grounds.

On the basis of the above, Ershov lays down the following requirements for a formal language to describe algorithms:

(1) The definition should comprise a rich class of elementary operations in contrast to the atomic character of the components of the theoretical formalisms.

(2) The definition should allow for the "instructions" of an algorithm to be developed in the course of its operation, as is the case with computer programs.

(3) The language should be flexible enough to reflect the features of particular classes of problems. For example, the problems of linear algebra should be described in terms of the field operations. Thus, the definition of an algorithm should allow both the form of the information to be processed and the means by which it is processed to vary with the class of algorithms considered.

Ersov is the author of "Programming Program for the BESM Computer" [6], defining a compiler based on the operator method of programming. These earlier efforts account for retention of the word "operator" in "operator algorithms," but there is another relationship:

> From the structural point of view the basic characteristic feature of logical schemes is that an algorithm is represented in the form of a list of information processing operators, all the transitions from one operator to another being indicated; the selection of one out of the several possible transitions, depending on the results obtained, is carried out by logical operators. This feature is retained entirely in our proposed definition of operator algorithms . . . .

3.3.2 The Definition of Operator Algorithms. The syntactic classes of Ershov's definition are those of variables, operations, formulas, and operators. In each case, the members of the classes are words on some alphabet. It is understood that this alphabet always includes the symbols $(,), \Rightarrow$, and *. For each class of operator algorithms, the class $\mathcal{V}$ of variables is a recursive subset of the set W of all words on the alphabet of the class. Variables are denoted by lowercase roman letters, with or without subscripts:

a, b, c, d, n, x, y, z, x_1, y_1, z_1, . . . etc.

The possible values of the variables also constitute a recursive subset of W. In general the specification of the set $\mathcal{V}$ of variables is to include mention of the possible values of the variables. Intuitively speaking, variables are to serve as the addresses of memory locations of a computer, and the values of these variables as the contents of the locations.

A k-place operation is a word of the form

$$A_0(\quad)A_1(\quad)\ldots(\quad)A_k,$$

where A_0, A_1, ..., A_k are words on the given alphabet. It is understood that the subalphabet of the alphabet of the operator algorithm in which the variables V and the words A_0, ..., A_k are written does not contain any of the symbols (,), $\Rightarrow$, and *. The result of applying $A_0(\quad)A_1(\quad)\ldots(\quad)A_k$ to words n_1, n_2, ..., n_k is written $A_0(n_1)A_1(n_2)\ldots(n_k)A_k$.

In presenting a class of operator algorithms, a finite list $\mathcal{L}$ of operations is given. The list corresponds to the "order code" of a computer, that is, to the selection of elementary operations with which a program is composed.

The notions of expression, arguments of expressions, and values of expressions for a given selection of argument values are simultaneously defined by induction:

(a) Any variable $x \in$ is an expression, and x is the argument of x. If a is a value of the variable x, a is a value of the expression x.

(b) Let $A_0(\quad)A_1(\quad)\ldots(\quad)A_k$ be a k-place operation of $\mathcal{L}$ and

$$T_1(x_{11}, x_{12}, \ldots, x_{in_1}), \ldots, T_k(x_{k1}, x_{k2}, \ldots, x_{kn_k})$$

be expressions. The word T obtained by substituting in the i-th place of the operation $A_0(\quad)A_1(\quad)\ldots(\quad)A_k$ the expression $T_i(x_{i1}, x_{i2}, \ldots, x_{in_i})$ $(i = 1, 2, \ldots, k)$, namely

$$A_0(T_1(x_{11}, \ldots, x_{1n_1}))\ldots(T_k(x_{k1}, \ldots, x_{kn_k}))A_k,$$

is an expression with arguments $x_1, x_2, \ldots, x_n$, where $x_1, x_2, \ldots, x_n$ constitute all those variables that are arguments of the expressions $T_1, \ldots, T_k$. Let $a_1, a_2, \ldots, a_n$ be the values of $x_1, \ldots, x_n$. Each a_ι $(\iota = 1, 2, \ldots, n)$ is some a_{ij} $(1 \leq i \leq k;\ 1 \leq j \leq n_i)$.

If $T_1(a_{11}, \ldots, a_{1n_1}) = b_1$, $T_2(a_{21}, \ldots, a_{2n_2}) = b_2$, $\ldots$, $T_k(a_{k1}, a_{k2}, \ldots, a_{kn_k}) = b_k$ and $A_0(b_1)\ldots(b_k)A_k = b$, the expression T has the value $T(a_1, \ldots, a_n)$, and $T(a_1, \ldots, a_n) = b$.

A formula is a word of the form

$$T(x_1, x_2, \ldots, x_n) \Rightarrow y,$$

where $\Rightarrow$ is the sign of correspondence, $T(x_1, \ldots, x_n)$ is

an expression, and y is a variable. The variables x_1, x_2, ..., x_n are called the input variables of the formula, and y is the output variable.

An operator is any word of the form

$$T_1 \Rightarrow y_1 * T_2 \Rightarrow y_2 * \ldots * T_m \Rightarrow y_m,$$

where $m > 0$, and $T_i \Rightarrow y_i$ $(i = 1, 2, \ldots, m)$ are formulas.

It is the operators that are the basic building blocks of operator algorithms. They are the formal analogue of the instructions of a computer program. In an operator algorithm, the operators occur as the values of certain variables, to be termed the operator variables. Thus, in any operator algorithm certain variables must be designated as the operator variables.

Suppose that classes $\mathcal{U}$ and $\mathcal{L}$ of variables and operations on some alphabet are given. A word n_0, distinct from all the variables, is set aside to identify a STOP; r + s variables $p_1, \ldots, p_r, x_1, \ldots, x_s$ of $\mathcal{U}$ are selected, the $p_1, \ldots, p_r$ serving as the parameters and the $x_1, x_2, \ldots, x_s$ as the functional variables of an operator algorithm. Associated with $p_1, \ldots, p_r$ are designated values $q_1, \ldots, q_r$. From among the parameters, one variable p* is selected as the start variable. Finally, a choice of functional variables $x_1, x_2, \ldots, x_s$ and parameters $p_1, \ldots, p_r$ with a start p* and values $q_1, \ldots, q_r^*$ is an operator algorithm

$\mathcal{A}$ from the class of operator algorithms $\mathcal{A}(\mathcal{V},\mathcal{L})$ on the variables $\mathcal{V}$ and operations $\mathcal{L}$.

3.3.3. The Execution of an Operator Algorithm.

The execution of an operator algorithm is to be conceived in terms of the operating region of the algorithm. This region consists of certain variables, together with the values of these variables. The effect of operation of the algorithm is to transform the operating region. As a result of this transformation, the values of some variables in the operating region are changed, and new variables, together with their values, are added to the operating region. The acts of the algorithm by which these changes are effected reduce to four: (a) transition to a variable, (b) execution of an operator, (c) normal stop, (d) stop without result. Of these, (b) is the most important.

A condition for the start of the algorithm $\mathcal{A}$ is that the function variables $x_1, x_2, \ldots, x_s$ be assigned the values $a_1, a_2, \ldots, a_s$, the initial data of the algorithm $\mathcal{A}$.

Initially, the operating region of $\mathcal{A}$ consists of the variables $x_1, x_2, \ldots x_s, p_1, \ldots, p_s$ and their values $a_1, \ldots, a_s, q_1, \ldots, q_s$.

Step 1. The start variable p* belongs to the operating region of $\mathcal{A}$.

Step k+1. Suppose that k steps have been performed and that the k-th step has resulted in a transition to the variable y in the operating region and the present value of y is b.

Case 1. b is not an operator. The process terminates in a stop without result.

Case 2. b is an operator $T_1 \Rightarrow y_1 * \cdots * T_m \Rightarrow y_m$. Let

$$T_i(z_1,\ldots,z_n) \Rightarrow y_i$$

be the i-th formula of the operator, i = 1,2,... n, with variables $z_1, z_2,\ldots, z_n$. If any of $z_1, z_2,\ldots, z_n$ is not included in the operating region, the procedure terminates in a stop without result. On the other hand, suppose all of $z_1,\ldots, z_n$ occur in the operating region with values $v_1, v_2,\ldots, v_n$. If $T_i(v_1,\ldots v_n)$ is undefined or $T_i(v_1,\ldots, v_n) = v$ and v is not an admissible value of y_i, the procedure terminates in a stop without result. If $T_i(v_1,\ldots, v_n) = b_i$, and y_i does not belong to the operating region, y_i is added to the region with assigned value b_i; if y_i already is a member of the region with value b_i*, then b_i* is replaced by b_i. This procedure is carried out successively for i = 1, 2,..., m. It is understood that if for any i, $y_i = y$, the replacement of b by b_i

is made after the procedure has been carried out for $i = 1, 2, \ldots, m$.

If b_m, the value of y_m, equals n_0, the procedure comes to a normal stop with the result being the values of all variables belonging to the operating region of $\mathcal{A}$. If $b_m \neq n_0$ and either b_m is not a variable or $b_m = y$ and y is not in the operating region, the procedure terminates in a stop without result. If $b_m = y$ and y is in the operating region, there occurs a transition to the variable y, and the next step is executed.

An algorithm $\mathcal{A}$ with parameters $p_1, \ldots, p_s$, functional variables $x_1, \ldots, x_s$, and initial data $d_1, \ldots, d_s$ is represented as

$$p_1 \rfloor q_1;\ p_2 \rfloor q_2; \ldots;\ p_r \rfloor q_r;\ x_1 \rfloor d_1; \ldots,\ x_s \rfloor d_s,$$

where it is understood that ";" and "$\rfloor$" are not members of the alphabets in which the variables or the values of the variables of the class of algorithms are written.

3.3.4 Examples.

1. Algorithms of the class $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$. The alphabet consists of the Arabic numerals. The numerals representing even numbers different from zero are called numerical variables. The values of the numerical variables are the natural numbers. The representations of the odd

numbers and 0 are called operator variables. Their values are words on the alphabet $\{0,\ldots,9, +, P, (,), *, \Rightarrow\}$.

The list $\mathcal{L}_1$ consists of the following two operations:

(a) $(\ ) = 1$, where $(\mu) + 1 = \mu + 1$;

(b) $P\ ((\ )(\ ))$, such that

$$P\ ((\mu_1).(\mu_2)) = \begin{cases} 1 \text{ if } \mu_1 \leq \mu_2, \\ 0 \text{ if } \mu_1 > \mu_2, \end{cases}$$

where μ, μ_1, and μ_2 are natural numbers.

An algorithm for addition:

$o \rfloor u \Rightarrow s;\ 1 \rfloor v \Rightarrow s;\ 3* \rfloor t \Rightarrow q*r \Rightarrow s;$

$5 \rfloor P(y,q) \Rightarrow s;\ 7 \rfloor x+1 \Rightarrow x*q+1 \Rightarrow q*r \Rightarrow s;$

$v \rfloor 2;\ t \rfloor 0;\ r \rfloor 5;\ \ u \rfloor 7;\ x \rfloor;\ y \rfloor .$

The only functional variables here are x and y.

Given initial data d_1, d_2, we will compute 2 + 1 as follows.

Step	Transition Variable	Operating Region
0		$R_o = \{0,1,3*,5,7,v,t,r,u,x,y\}$
1	3*	$R_1 = R_o$
2	5	$R_2 = R_o \cup \{q,s\}$
3	0	$R_3 = R_2$
4	7	$R_4 = R_2$
5	5	$R_5 = R_2$
6	1	$R_6 = R_2$

Result = {u ⇒ s, v ⇒ s, t ⇒ q*r ⇒ s, P(y,q) ⇒ s,

x+1 ⇒ x*q+1 ⇒ q*r ⇒ s,2,0,5,7,3,1,1,2}.

An algorithm of class $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$ for multiplication

0 ⌋u ⇒ s; 1 ⌋v ⇒ s; 3* ⌋t_0 ⇒ z*t_0 ⇒ k*t_5 ⇒ s;

5 ⌋(t_5+1)+1 ⇒ u*t_2 ⇒ v*P(y,k) ⇒ s; 7 ⌋t_0 ⇒ p*k+1 ⇒ k

((((t_5+1)+1)+1)+1 ⇒ s; 9 ⌋(s+1)+1 ⇒ ut_5 ⇒ v*P(x,p) ⇒ s;

11 ⌋z+1 ⇒ z*p+1 ⇒ p*(((t_5+1)+1 ⇒ s; t_0 ⌋0, t_2 ⌋2;

t_4 ⌋4; t_5 ⌋5; x⌋ ; y⌋ .

For any initial data d_1, d_2, the value of the variable z after application of this algorithm is $d_1 \cdot d_2$.

2. The class $\mathcal{A}(\mathcal{V}_2, \mathcal{L}_2)$. The variables $\mathcal{V}_2$ are words on the alphabet {|}, with the proviso that the word || signifies STOP. Words of length n on {|} are abbreviated as $|^n$; and $|^0$, as well as Λ, represents the empty word. The values of the variables are words on the alphabet A = {a_1,..., a_t}, where A indicates T,R,S, ,(,),*,⇒. The list of operations $\mathcal{L}_2$ consists of 2t+1 operations, where t is the number of letters in A.

Let P be an arbitrary word on A.

(a) The operation T( ), such that

$$T(P) = \begin{cases} P' \text{ if } P = \xi P'(\xi \in A), \\ \Lambda \text{ if } P = \Lambda. \end{cases}$$

(b) t operations of the form $R|^i( ), i=1,\ldots,t$

such that

$$R|^{i}(P) = a_i \cdot P$$

(c) t operations of the form $S|^{i}(\ )$, $i = 1, \ldots, t$, such that

$$S|^{i}(P) = \begin{cases} \wedge & \text{if } P = a_i \cdot P', \\ \wedge & \text{if } P \neq a_i \cdot P'. \end{cases}$$

A noteworthy difference between the algorithms of classes $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$ and $\mathcal{A}(\mathcal{V}_2, \mathcal{L}_2)$ is this: those of $\mathcal{A}(\mathcal{V}_2, \mathcal{L}_2)$ have the capability of generating operators while the only transformations of operators that are performed by those of $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$ are the relatively simple ones of the form $x \Rightarrow y$.

Here is one algorithm of the class $\mathcal{A}(\mathcal{V}_2, \mathcal{L}_2)$ that inverts a nonempty word and sends the empty word into itself. The following abbreviations are employed:

$S_{a_i}(\ )$ stands for $S|^{i}(\ \ )$,

$R_{a_i}(\ )$ stands for $R|^{i}(\ \ )$.

If $\xi_1 \ldots \xi_n$ is a nonnull word on A,

$R_{\xi_1 \ldots \xi_n}(x)$ stands for $R_{\xi_1}(R_{\xi_2}(\ldots(R_{\xi_n}(x))\ldots))$,

$T^{n}(x)$ stands for $\underbrace{T(T(\ldots T}_{n\ T\text{'s}}(x)\ldots))$

The algorithm:

$|^0 \lrcorner |^u \Rightarrow s;\ | \lrcorner |^v \Rightarrow s;\ |^3 * \lrcorner z_0 \Rightarrow y*z_4 \Rightarrow s;\ |^4 \lrcorner r_1 \Rightarrow k*r_2 \Rightarrow |^5 *r_3 \Rightarrow |^6 *z_5 \Rightarrow s;\ |^5 \lrcorner |^{\ell} \Rightarrow |^u * |^m \Rightarrow |^v * S|(x) \Rightarrow s;$

$|^6 \lrcorner R|(y) \Rightarrow y*T(x) \Rightarrow x*z_4 \Rightarrow s;\ |^7 \lrcorner T^{(\ell+u+m+v+5)}(|^5) \Rightarrow |^5 *R|^{\ell} \Rightarrow |^u * |^m \Rightarrow |^v * S|(|^5) \Rightarrow |^5 * T(|^6) \Rightarrow |^6 * R_{R|}(|^6) \Rightarrow |^6 * T(k) \Rightarrow k *z_5 \Rightarrow |^u * z_2 \Rightarrow |^v * S_a(k) \Rightarrow s;\ |^{\ell} \lrcorner |^7;\ |^m \lrcorner |^6;\ r_1 \lrcorner |^t a \quad (\in A,\ a \neq |)$

$r_2 \lrcorner |^{\ell} \Rightarrow |^u * |^m \Rightarrow |^v * S|(x) \Rightarrow s;\ r_3 \lrcorner R|(y) \Rightarrow y*T(x) \Rightarrow x*z_4 \Rightarrow s;$

$z_0 \lrcorner \Lambda;\ z_2 \lrcorner |^2;\ z_4 \lrcorner |^4;\ z_5 \lrcorner |^5;\ x \lrcorner .$

3.3.5. The Value and S-Representation of an Operator Algorithm. Ershov, in the manner of Yanov, takes the value of an algorithm, given some initial data, to be the sequence of operators performed in order of execution; that is, if algorithm $\mathcal{A}$ is applicable to initial data $d_1,\ldots,d_s$, and there results the execution of operators $n_1,\ldots,n_k$, the word $n_1*n_2*\ldots*n_k$ is the value of $\mathcal{A}$ for $d_1,d_2,\ldots,d_s$.

The value of the algorithm for addition of $\mathcal{A}(\mathcal{V}_1,\mathcal{L}_1)$ for $y = 3$, arbitrary x is

$t \Rightarrow q*r \Rightarrow s*P(y,q) \Rightarrow s*u \Rightarrow s*x+1 \Rightarrow x*q+1 \Rightarrow q$

$*r \Rightarrow s*P(y,q) \Rightarrow s*u \Rightarrow s*x+1 \Rightarrow x*q+1 \Rightarrow q*$

$r \Rightarrow s*P(y,q) \Rightarrow s*u \Rightarrow s*x+1 \Rightarrow x*q+1 \Rightarrow q*$

$r \Rightarrow s*P(y,q) \Rightarrow s*v \Rightarrow s.$

Suppose an operator algorithm $\mathcal{A}$ from the class $\mathcal{A}(\mathcal{V},\mathcal{D})$ with functional variables $x_1,\ldots,x_s$ is given. By the function realized by $\mathcal{A}$ relative to $y \in \mathcal{V}$ is meant

the function $y_{\mathcal{A}}(x_1,\ldots,x_s)$ defined for exactly those s tuples $(d_1,\ldots,d_s)$ to which $\mathcal{A}$ is applicable and such that when defined $y_{\mathcal{A}}(d_1,\ldots,d_s)=b$, where b is the value of y at completion of the application.

A distinctive feature of Ershov's treatment is his notion of the S—representation of a value of an operator algorithm. The motivation behind this notion seems to be as follows. In running a program on a computer it frequently happens that we are interested in certain intermediate results obtained in arriving at the "answer." On the other hand, much of the information processed in such a run is of little or no interest. Correspondingly, certain of the variables present in the value of an operator algorithm play an essential role in the calculation, while others play a merely transitional role. In defining the S—representation of a value, one seeks to isolate the more important variables.

Let Z be a value of algorithm $\mathcal{A}$, and let $F(x_1,x_2,\ldots,x_n) \Rightarrow y$ be the leftmost formula such that its output variable y is the input variable for another formula of Z to the right of $F \Rightarrow y$. Let Z' be that part of Z to the right of $F \Rightarrow y$ and Z_y that part of Z' bounded on the right inclusively by the first formula of Z to the right of $F \Rightarrow y$ which has y as an output variable. To obtain

the word $Z^{(1)}$ from $Z^0 = Z$, all occurrences of y as an input variable in formulas of Z_y are replaced by $F(x_1,x_2,\ldots,x_n)$ and if $Z' \neq Z_y$, the leftmost $F(x_1,\ldots,x_n) \Rightarrow y*$ is deleted from Z. This procedure is repeated until a word $Z^{(m)}$ is arrived at in which <u>all</u> formulas of the form $F \Rightarrow y$, where y is an input variable of some formula to the right of $F \Rightarrow y$, have been eliminated. Finally, all formulas of $Z^{(m)}$ having the same output variable except the rightmost are deleted and the result is the S—representation, S_Z, of Z. For each variable y occurring in S_Z, there is exactly one formula of the form $F(p_{i_1},\ldots,p_{i_\ell},x_{j_1},\ldots,x_{j_m}) \Rightarrow y$ where $p_{i_1},\ldots,p_{i_\ell}$, $x_{j_1},\ldots,x_{j_m}$ occur among the parameters and functional variables of algorithm $\mathcal{A}$.

If Z is a value of algorithm $\mathcal{A}$, all variables occurring as output variables in formulas of Z are called <u>resulting variables</u>. A formula $F \Rightarrow y$ of S_Z is called the <u>S—representation</u> of the resulting variable y.

The function of the S—representation of a variable is brought out in the following.

<u>THEOREM</u>. <u>Let</u> $F(p_{i_1},\ldots,p_{i_\ell},x_{j_1},\ldots,x_{j_m}) \Rightarrow y$ <u>be an S- representation of the resulting variable</u> y <u>for a value</u> Z <u>of algorithm</u> $\mathcal{A}$, <u>given initial data</u> $d_1,\ldots,d_s$. <u>Then if</u> $y_{\mathcal{A}}(x_1,\ldots,x_s)$ <u>is a function realized by</u> $\mathcal{A}$.

$$y_{\mathcal{A}}(d_1,\ldots,d_s) = F(q_{i_1},\ldots,q_{i_\ell},d_{j_1},\ldots,d_{j_m})$$

where $q_{r_1},\ldots,q_{i_\ell}$ are the initial values of the parameters $p_{i_1},\ldots,p_{i_\ell}$ of $\mathcal{A}$.

The S-representation of the value of the above algorithm for addition when y = 3 is

$S_Z = (((x) + 1) + 1 \Rightarrow x*(((t) + 1) + 1) + 1) \Rightarrow q*v \Rightarrow s.$

3.3.6 Operator Algorithms and Graph Schemes. Consider a class $\mathcal{A}(\mathcal{V},\mathcal{L})$ of operator algorithms such that among the operations of $\mathcal{A}$ there is at least one test operation, which upon application results in one of two operator variables, n^+ or n^-.

A class of graph schemes $\mathcal{S}(\mathcal{V},\mathcal{L})$ is associated with $\mathcal{A}(\mathcal{V},\mathcal{L})$ in the following manner. The operators of $\mathcal{A}(\mathcal{V},\mathcal{L})$ defined over $\mathcal{V}$ and $\mathcal{L}$, plus a null operator, constitute the operators of the graph scheme. Those formulas $T \Rightarrow y$ such that T is a test operation to be applied to expressions serve as the discriminators of $\mathcal{L}$. To complete the definition, it is stipulated that associated with each member of $\mathcal{S}(\mathcal{V},\mathcal{L})$ is certain initial information, consisting of parameters $p_1,\ldots,p_r$ with assigned respective values $q_1,\ldots,q_r$ and functional variables $x_1,\ldots,x_s$.

To relate the execution of a graph scheme of $\mathcal{S}(\mathcal{V},\mathcal{L})$ to that of an algorithm of $\mathcal{A}(\mathcal{V},\mathcal{L})$ note that the result

of applying a discriminator of the former is understood to be a plus if application of the corresponding operation of the latter has resulted in n^+, and minus if application has resulted in n^-. The syntactic definitions of operating region, value of a graph scheme, the S—representation of resulting variables, the functions realized by a graph scheme, etc., can now be given. To complete the comparison, we define an algorithm $\mathcal{A}$ to be of zero rank if all operator variables of $\mathcal{A}$ occur among the parameters and if upon application of $\mathcal{A}$ the values of the operator variables remain unchanged throughout the application.

For example, a graph scheme of the previously given algorithm in $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$ for multiplication is shown in Fig. 1. the following figure.

After these lengthy preliminaries, Ershov proves the following

THEOREM. For any graph scheme $\mathcal{G} \in \mathcal{G}(\mathcal{V}, \mathcal{L})$ with functional variables $x_1, \ldots, x_s$, there is an operator algorithm of zero rank $\mathcal{A} \in \mathcal{A}(\mathcal{V}, \mathcal{L})$ with the same functional variables such that for the set of initial data $d_1, d_2, \ldots, d_s$ if y is the resulting variable of the graph scheme $\mathcal{G}$ and has an S—representation $F_{\mathcal{G}} \Rightarrow y$, then y is a resulting variable of $\mathcal{A}$ and for the same set of initial data has an S—representation $F_{\mathcal{A}} \Rightarrow y$ such that

$$F_{\mathcal{A}} = F_{\mathcal{G}}.$$

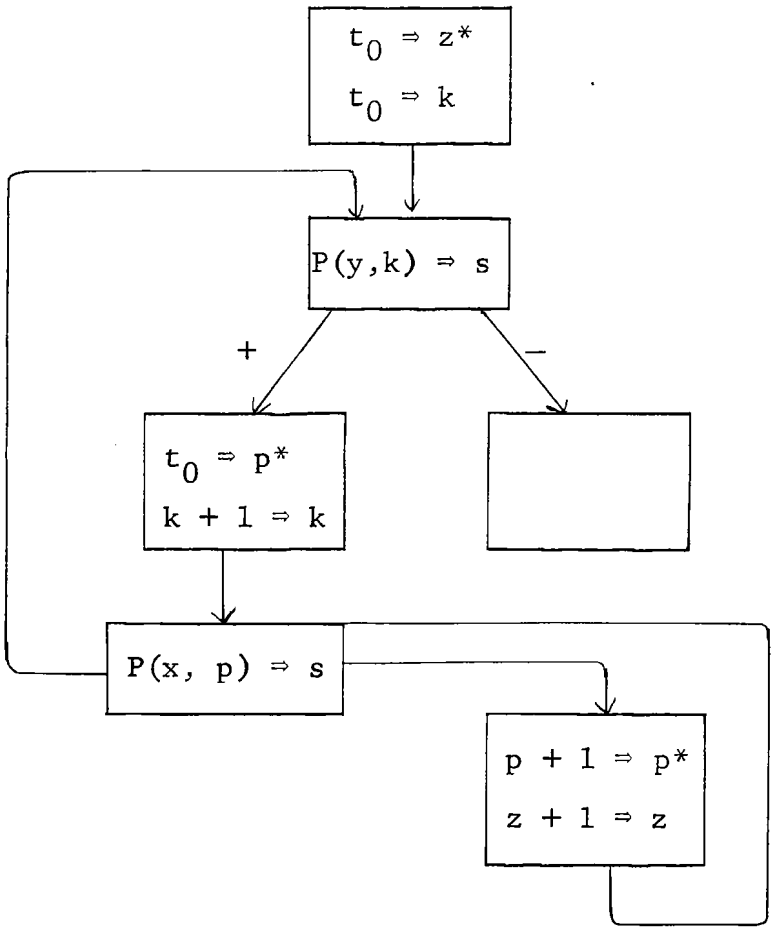

$t_0 \Rightarrow z^*$
$t_0 \Rightarrow k$
P(y,k) ⇒ s
+
−
$t_0 \Rightarrow p^*$
k + 1 ⇒ k
P(x, p) ⇒ s
p + 1 ⇒ p*
z + 1 ⇒ z

Fig. 1

By use of this theorem and much bookkeeping he proves:

(1) A function φ is partial recursive if and only if it is realized by an algorithm of $\mathcal{A}^{0}(\mathcal{V}_1, \mathcal{L}_1)$ (actually Ershov proves a weakened version of the "if" part of this theorem). Here $\mathcal{A}^{0}(\mathcal{V}_1, \mathcal{L}_1)$ is the subclass of $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$ consisting of algorithms of zero rank.

(2) For any normal algorithm $\mathcal{A}$ in alphabet $A = \{a_1, \ldots, a_t\}$ where $\rightarrow$ and $\cdot$ are not in A, there is an algorithm $\mathcal{A}^{*}$ of $\mathcal{A}(\mathcal{V}_2, \mathcal{L}_2)$ with one functional variable x and with a resultant variable y such that

$$\mathcal{A}(x) \simeq y_{\mathcal{A}^{*}(x)},$$

where $\simeq$ denotes strong equality and $\mathcal{A}(x)$ is the word function defined by $\mathcal{A}$ such that $\mathcal{A}(P) = Q$ is the result of applying $\mathcal{A}$ to P, where P and Q are words on A.

Two points one notices about Ershov's reformulation of the notion of algorithm are its breadth and elegance. His efforts are very reminiscent of the work of Wang [34], Minsky [25], and Shepherdson and Sturgis [32], in which the theory of Turing machines is redone to effect a rapprochement between theory and practice. In part, however, these later papers are concerned with demonstrating that

various combinations of rather special logical operations and arithmetic functions suffice for the computation of all partial recursive functions. To put it another way, it is shown that some machines obtained by crippling and/or modifying a Turing machine in some peculiar manner are universal. For example, in the paper of Shepherdson and Sturgis [32], one aim is to modify the definition of Turing machine to arrive at a machine for which intuitive computational procedures are easily programmed. These authors, however, also define a graded series of machines obtained by restricting the operation and changing the definition of their initial URM (Unlimited Register Machine) in various ways. They prove such results as the fact that a machine using a single binary tape and having two control heads--a right-hand head that can write, move only to the right, and print only when moving, and a left-hand head that can also move only rightwards and read only when moving (possibly destroying whatever it reads)--is a universal machine. Ershov's paper seems to have a more unified aim--applications to the theory of programming and, in particular, to the development of programming languages. It does take a while before one gets accustomed to his formalism and is able to carry out computations without

delay; indeed, the format of the papers [25], [32], and [34] resembles much more closely actual computer programs, the basic instructions being chosen and named to mirror actual machine operations. For example, a verbal translation of the basic instructions of the URM of [32] is as follows: (a) add 1 to the number in register n; (b) subtract, from the number in register n; (c) clear register n; (d) copy from register m into register n; (e) jump to exit 1; (f) jump to exit 1 if register m is empty. In a word, Ershov's treatment is more abstract. This is possible because of his careful analysis of the requirements of a formal language to describe algorithms and the way in which those features are built into his definition of operator algorithm.

3.3.7. The Amended Definition. In [11], Ershov shows how to represent computer programs and the logical schemes for programs as operator algorithms. He amends the definition of operator algorithm in two ways, one for purely technical convenience, the other for weightier reasons.

First, a zero-place operation is defined to be one of the form A where the word ( ) is not a subword of A. If A is a 0-place operation, an occurrence of an expression of the form A ⇒ y amounts to giving the variable y some one value. The point of introducing 0-place operations is that they

reduce the number of parameters required in an operator algorithm.

The second change has to do with the representation of an operator. An operator is a word of a definite form, namely,

$$T_1 \Rightarrow y_1 * T_2 \Rightarrow y_2 * \cdots * T_m \Rightarrow y_m,$$

where $T_i \Rightarrow y_i$ for $i = 1,2,\ldots,m$, are formulas. They occur in an algorithm as the values of operator variables. The fact that the values of the operator variables must be of one form introduces a rigidity into the definition that can be disadvantageous. For example, in the case of algorithms of the class $\mathcal{A}(\mathcal{V}_1, \mathcal{L}_1)$, the only possible transformations of the operator variables are those of the form $x \Rightarrow y$. These considerations argue for including a decoding operation in the definition of operator algorithm that would allow the values of operator variables to assume a form suited to the given class of algorithms and be subsequently transformed into an operator upon application of the decoding operation. Ershov maintains that this arrangement is consistent with computing practice in that a binary representation may serve as either a number or an instruction, depending on whether it appears in an arithmetic or control register.

The introduction of the decoding operation into the definition of operator algorithm necessitates an additional clause in the definition of the application of an algorithm $\mathcal{A}$ to initial data $d_1, d_2, \ldots, d_s$., as follows:

Suppose y is the current transition variable and is a member of the operating region R of $\mathcal{A}$ and has the current value b'. If the decoding operation is not defined on b', the process terminates in a stop without result. If when applied to b' the operation yields b, and b is not an operator, the process terminates in a stop without result; otherwise, the process continues as previously defined.

3.3.8 The Equivalence of Operator Algorithms. Ershov in conclusion turns to a discussion of the equivalence of algorithms, noting both the theoretical and practical importance of this problem. The measure of the strength of a definition of equivalence is the manner in which the "output" of an algorithm is defined. If equivalence of algorithms is taken to mean equal, or isomorphic, outputs for equal, or isomorphic, inputs, where "output" means the final result in the usual sense, then the general problem of deciding equivalence is recursively unsolvable. If it means that for all admissible inputs two algorithms produce the same total sequence of operations in the same order,

then the problem is recursively solvable, as shown by Yanov. Ershov feels, however, that Yanov's definition is too strong and that it excludes many algorithms that ought to be regarded as equivalent. He proposes a definition somewhere between Yanov's and the general one, based on the notion of the S-representation of the value of an algorithm, reasoning that

> a definition of equivalence based on the use of the S-representation for the output [enables one] ... to say that two algorithms are equivalent with respect to specified variables when corresponding variables in coinciding inputs are calculated by the same formulas. This is so, inasmuch as the S-representation of a variable is nothing other than an explicit expression of the formulas with which the resulting value of the variable is calculated.
>
> The most important implication of what we have said is that in programming (especially at the transition from logical schemes to the machine program), obviously, the most important role is played by transformations of algorithms which leave the calculation formulas (i.e., the S-representations) intact. Thus, such basic programming devices as division of the whole problem into parts and assigning the parts to different sections of the machine's memory, separation of subroutines, partition of formulas and identification of various intermediate results, transformation of logical operators, economizing on commands and working cells--all these lead to transformations of algorithms which preserve the S-representation of the resulting variable.

Consider now operator algorithms a_1 and a_2 of the same class of algorithms.

Certain variables in each are distinguished by the fact that they, as a result of applying the algorithm, take the really material values. It is assumed that there is

a 1-1 correspondence between these variables, as well as between the functional variables and those parameters in the algorithms that can enter into the S-representations of the distinguished variables. Let the functional variables, parameters, and distinguished variables of $\mathcal{A}_1$ be, respectively, $x_1,\ldots,x_s$, $p_1,\ldots,p_r$, and $y_i,\ldots,y_n$, and the corresponding variables of $\mathcal{A}_2$ be $\bar{x}_1,\ldots,\bar{x}_s$, $\bar{p}_1,\ldots,\bar{p}_r$, and $\bar{y}_1,\ldots,\bar{y}_n$.

$\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be equivalent with respect to the distinguished variables $\bar{y}_1,\ldots,\bar{y}_n$ and $\bar{y}_1,\ldots,\bar{y}_n$ if for all initial data $d_1,\ldots,d_s$ wherever one of the algorithms, say $\mathcal{A}_1$, is applicable to $d_1,\ldots,d_s$ and the S-representations of the distinguished variables have the form

$$T_1(x_{i_{11}},\ldots,x_{i_1 m_1},p_{j_{11}},\ldots,p_{j_1 \ell_1}) \Rightarrow y_1$$

$$T_2(x_{i_{21}},\ldots,x_{i_2 m_2},p_{j_{21}},\ldots,p_{j_2 \ell_2}) \Rightarrow y_2$$

$$\vdots$$

$$T_n(x_{i_{n1}},\ldots,x_{i_{nm_n}},p_{j_{n_1}},\ldots,p_{j_{n\ell_n}}) \Rightarrow y_n$$

then the other, say $\mathcal{A}_2$, is applicable to $d_1,\ldots,d_2$ and the S-representations of the distinguished variables have the form

$$T_1(\bar{x}_{i_{11}},\ldots,\bar{x}_{i_1 m_1},\bar{p}_{j_{11}},\ldots,\bar{p}_{j_1 \ell_1}) \Rightarrow \bar{y}_1$$

$$T_2(\bar{x}_{i_{11}},\ldots,\bar{x}_{i_2 m_2},\bar{p}_{j_{21}},\ldots,\bar{p}_{j_2 \ell_2}) \Rightarrow \bar{y}_2$$

$$\vdots$$

$$T_n(\bar{x}_{i_{n1}},\ldots,\bar{x}_{i_{nm_n}},\bar{p}_{j_{n_1}},\ldots,\bar{p}_{j_{n\ell_n}}) \Rightarrow \bar{y}_n$$

For example, consider the algorithm for addition $\mathcal{A}'(\mathcal{V}_1,\mathcal{L}_1)$:

1⌋u ⇒ s; o⌋v ⇒ s; 3*⌋t ⇒ q*r ⇒ s; 5⌋P(q,x) ⇒ s;

7⌋ y+1 ⇒ y*q+1 ⇒ q*r ⇒ s;

v⌋2; t⌋1; r⌋5; u⌋7; y⌋; x⌋.

Let φ define the following correspondence between the functional variables, parameters, and specified variables of $\mathcal{A}'(\mathcal{V}_1,\mathcal{L}_1)$ and the previously defined $\mathcal{A}(\mathcal{V}_1,\mathcal{L}_1)$ as follows:

$\varphi(0) = 1$, $\varphi(1) = 0$, $\varphi(3) = 3$, $\varphi(5) = 5$, $\varphi(7) = 7$,

$\varphi(v) = v$, $\varphi(t) = t$, $\varphi(r) = r$, $\varphi(u) = u$, $\varphi(x) = y$,

$\varphi(y) = x$.

$\mathcal{A}'(\mathcal{V}_1,\mathcal{L}_1)$ and $\mathcal{A}(\mathcal{V}_1,\mathcal{L}_1)$ are equivalent with respect to y and x in the sense of Ershov, as the reader can readily check.

To cover the case in which $\mathcal{A}_1$ and $\mathcal{A}_2$ belong to

different classes of operator algorithms then there are the following additional requirement: there is an isomorphism between the sets of operations $\mathcal{L}_1$ and $\mathcal{L}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, and between the possible values of the variables in the sets $\mathcal{V}_1$ and $\mathcal{V}_2$. Under these conditions, $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be <u>equivalent</u> if when applied to isomorphic initial data, they result in S-representations of the distinguished variables that coincide up to isomorphism of the operations generating them.

The bulk of [11] is devoted to defining two classes $\mathcal{A}(\mathcal{V}_{CC}, \mathcal{L}_{CC})$ and $\mathcal{A}(\mathcal{V}_{LS}, \mathcal{L}_{LS})$ of operator algorithms. The former describes the programs written in machine language for the conventional computer (CC) defined in Lyapunov's [21], whereas the latter describes variants of the logical schemes used as input information to the PPS, the compiler for the Strela machine. Ershov defines an algorithm from each of these classes to solve a pair of linear equations and shows these algorithms to be equivalent. The details of the definitions of $\mathcal{A}(\mathcal{V}_{CC}, \mathcal{L}_{CC})$ and $\mathcal{A}(\mathcal{V}_{LS}, \mathcal{L}_{LS})$ are extensive, and the reader is referred to [11].

4. A Comment

An editorial comment concerning the style of many of the papers surveyed is in order. In general, the style is ponderous, employing much symbolism. The dry, atomistic analysis will be egregiously familiar to those who have read papers in recursive function theory by Russian authors, particularly those in recursive analysis. A special, trying characteristic of some of the papers--of [21], [27], and [19], for example--is the encyclopedic recapitulation of the genesis and nature of programming.

The lengthy, careful preambles are in this case symptomatic of the seriousness of the whole effort. The paper [21] of Lyapunov initiated a systematic, serious attempt to make computer programming a mathematical discipline. The semiformal nature of the operator method perpetuates the view of programming as a several-stage process. Indeed, Lyapunov emphasizes the distinction between a calculation scheme and a program scheme. The calculation scheme should be the result of the analysis of the algorithm into its major parts, and is the symbolic expression of the algorithm as the combination of these parts. As such it does not reflect any of the characteristics of a computing machine. The expansion of the abstract operators of the calculation

scheme as the product of various kinds of control operators, arithmetical operators, and connecting logical conditions results in the program scheme, which does reflect some basic characteristics of a computing machine. For example, the control operators direct that the memory of the machine be used in such a way as to effect successive execution of the arithmetical operators. Prior to coding the program scheme in machine language, the programmer may seek to improve the scheme in order to produce a more efficient program by applying various kinds of transformations. Among these transformations are those closely related to the work of Yanov, namely, transformations of the logical conditions present in a program scheme. Indeed, standard Soviet texts on computing machines (such as [20]) include chapters on the "formal transformations of the logical schemes of programs." It is clear that learning to program in this fashion is a more arduous task than learning to write acceptable programs in a language such as FORTRAN.

The most convincing evidence of the seriousness of the Soviet effort is the substantial identity of those who are responsible for the more important uses of the operator method and those who have developed the theory surveyed in this study. Thus, if one looks into Soviet work on automatic

programming, he frequently reads about the contributions of A. A. Lyapunov, his pupils, and collaborators; and, indeed, the mathematical theory herein surveyed is due to them. Turski writes in [33]: "It seems that apart from several interesting exceptions...all major work on automatic programming in the USSR is done in notation, and uses concepts and definitions conceived by Lyapunov's team." Again, A. P. Ershov is the author not only of [10], [11], and [12], but also of [6], [8], and [9]. It is well to observe here that this commitment to the development of a mathematical theory of programming on the part of investigators who were also greatly interested in practical matters did not soon cease: Lyapunov not only began the investigation in [21], but also initiated a more abstract, algebraic treatment in [22].

The most important use of operator methods in the Soviet Union has been for the automation of programming. By contemporary Western standards much of the labor associated with use of a PP must appear primitive in nature. The already toilsome task of composition of program schemes in manual programming becomes yet more taxing when the program is to be composed with the assistance of a PP. In manual programming the program scheme is an aid. It serves as a guide to the writing of the program in machine language, and hence it does not always need to be constructed in full detail. But a program scheme used as an input to a

PP represents the programmer's final product. It is the fruit of his labor. Thus use of PPs further raised the technical nature of a programmer's work. (In contrast, in the United States at about the same time, automatic programming languages such as FORTRAN were being introduced to enable virtually anyone having a moderately technical background to learn to be a programmer.) Moreover, until quite recently no Soviet-made computer had a complete alphanumeric input-output system. In fact, alphanumeric key punchers have been scarce. Thus, generally, coding has had to be done in machine language encoded in digits prior to key punching. The machine coding of an elaborate program scheme increases the chance of the programmer committing miniscule errors that are hard to detect, but ruinous.

The method of PPs has not met with unanimous acceptance within the Soviet Union. Criticism and an alternate approach are given briefly in [16]. The PP method has, however, been one of the principal approaches. Currently work is being concentrated on various modifications and extensions of ALGOL [1], [8], [7]. The Soviets regard the introduction of ALGOL as a major pioneering step in the development of automatic programming systems. For example, Ershov states in [12]:

> with respect to the language ALGOL 60, it should be recognized that with its appearance there ensued a new stage in the development of the theory and automation of programming and there

> emerged new levels of ideas and methods. In the first place, with the appearance of ALGOL [redacted] programming really became an international discipline, not only because ALGOL is used as a language for expressing algorithms, but also due to the increase in the general meaning of results connected with the realization of ALGOL. Besides, the richness of its means of representation, the comparative difficulty of its realization, the general logical level of the language, in particular the definition of its syntax have sharply increased the role of theoretical research in the design of translators.

Programming languages such as ALGOL were developed in the West and were designed and implemented without relying on any theory comparable to that deriving from the operator method. Thus, the major question concerning the mathematical theory surveyed herein is: If the means of programming and, in particular, the means of automatic programming that have been so closely allied in authorship and direction to the theory surveyed have shifted to the use of higher level languages utterly independent of the theory, is the development of the theory to be further pursued, and, if so, to what end?

The theory has had some effect on automatic programming systems. The Yanov notation is used in defining value assignments of variables in [16]. Also, it is plausible that the transformations of Yanov are being used in the Alpha-translator of the Alpha automatic programming system developed by A. P. Ershov and his colleagues at Novosibirsk. In [12], Ershov gives a transcription of the transformations of Yanov in terms of graph schemes and presents the system

in connection with the algebraic problems arising in construction of a translator. In [9], he says in reference to the Alpha translator: "The main methods of improving the efficiency of programming are the applications of formal transformations and a mixed strategy of programming--all this as the basis of multiphase translation." In describing this multiphase translation, he states that the "work of the translation proper goes from the highest level language into the lowest one and is followed by a series of a formal transformation on the lowest level language, oriented toward the optimization of the object program." He proceeds to list five catagories of these optimizing transformations, and at least some of these categories could make use of Yanov's work--for example, the category of eliminating redundant expressions in unbranched parts of an Alpha program.

As for future practical applications of the theory, some authors expect big things to come. For example, Turski in [33] affirms: "There is no doubt when alphanumerical devices become widely available for Soviet-made computers, the tremendous theoretical work done in that country will ripen into many interesting automatized programming systems."

Irrespective of whether this has been borne out in the Alpha programming system, we think that this statement is probably unduly optimistic, at least insofar as the later theory surveyed in this report is concerned. Thus, it may well be that Lyapunov in confidently noting the strong similarity between the transformations that occur in computer programming and those of the theory of categories has exaggerated the situation. It would be foolish, however, to rule out the possibility of the practical relevance of this work. In this connection, the question arises as to the possible use in automatic programming of the work of Glebov on the synthesis of operators of a given class from measurably simpler operators of that class.

It is also quite likely that Ershov's reformulation of the notion of algorithm was of use to him in his work on automatic programming languages. His latest theoretical efforts are, however, somewhat puzzling. After in [11] criticizing Yanov's definition of the equivalence of two algorithms as being too narrow to be of practical value and proposing an alternate presumably wider definition, he in [13], so far as can be adduced from [12], works with Yanov's definition, and derives Yanov's equivalence theorem

in a graph-theoretic formalism. What is the reason for this apparent backsliding? Has any research been conducted connected with decision problems relative to his own definition?

The most reliable answers to these questions should probably be obtained from the authors of the papers reviewed. Otherwise, we will just have to wait and see if the questions are resolved by the contents of future publications.

REFERENCES

1. Ageev, M. I., "Ocnovy Algoritimicheskogo Yazyka Algol–60," Bsychislitelnyj tsentr An SSSR ("The Basic Algorithmic Language Algol–60," Computing Center of the Academy of Sciences of the USSR), Moskva, 1965.

2. Arsenteva, N. G., "Ob Nekotorykh Preobrazovani Yakh Skhem Programm," Problemi Kibernetiki, Vol. 4, Gocy–darstvennoe Izdalelstvo Fizico–Matematicheskoj Literatury, Fizmatgiz, Moskva, 1958, pp. 59–68. Translation: "Some Transformations of Program Schemes," Problems of Cybernetics, Vol. 4, Pergamon Press, New York, 1962, pp. 1201-1211.

3. Birkhoff, G., Lattice Theory, revised edition, American Mathematical Society Colloquium Publications, Vol. XXV, American Mathematical Society, New York, 1948.

4. Daugvet, O. K., I. V. Klokachev, and L. A. Pykhovets, "Ob Automatizatsii Programmirovaniya," Leningrad Engineering–Economics Institute Imeni Palmiro Togliatti, Vol. 98, pp. 262–267.

5. Davis, M., Computability and Unsolvability, McGraw–Hill Book Company, Inc., New York, 1958.

6. Ershov, A. P., Programming Program for the BESM Computer, translated by A. Nadler, edited by J. P. Cleave, Pergamon Press, New York, 1959.

7. Ershov, A. P., G. I. Kozhukhin, U. M. Voloshin, Input Language for Automatic Programming Systems, translated with Introduction by R. W. Hockney, Academic Press, New York, 1963.

8. Ershov, A. P., et al, Alpha Sistem Avtomatizatsii Programmirovaniya (The Alpha System of Automatic Programming), Novosibirsk, 1965.

9. Ershov, A. P., "Alpha—An Automatic Programming System of High Efficiency," Journal of the Association of Computing Machinery, Vol. 3, No. 1, 1956, pp. 17-24.

10. Ershov, A. P., "Operatorny Algorifmy I," Problemi Kibernetiki, Vol. 3, Fizmatgiz, Moskva, 1960. Translation: "Operator Algorithms I," Problems of Cybernetics, Vol. 3, Pergamon Press, New York, 1962, pp. 697–763.

11. Ershov, A. P., "Operatorny Algorifmy II (Opiranie Osnovnykh Konstryktsij Programmirovaniya)," Problemi Kibernetiki, Vol. 8, Fizmatgiz, 1962, pp. 211–234. Translation:

"Operator Algorithms II (Basic Programming Constructions)," Problems of Cybernetics, Joint Publications Research Service, Department of Commerce, Washington, D.C., 1964, pp. 377–407.

12. Ershov. A. P., "Nekotorpye Voprosy Teorii Programmirovaniya i Konstryirovaniya Translyatorov" ("Some Questions in the Theory of Programming and the Construction of a Translator"), Abtorefevat Dissertatsii, Novosibirsk, 1966.

13. Ershov, A. P., "Operatorny Algoritmy III (Ob Operatonykh Skhemakh Yannova)," Problemi Kibernetiki (v pechati). Translation: "Operator Algorithms III (On the Operator Schemes of Yanov)," Problems of Cybernetics (in press).

14. Glebov, N. I., "Sintez Operatorov," Problemi Kibernetiki, Vol. 8, Fizmatgiz, Moskva, 1962. Translation: "Synthesis of Operators," Problems of Cybernetics, Joint Publications Research Service, Department of Commerce, Washington, D.C., 1964, pp. 191–200.

15. Glebov, N I., "Ob Algebraickeskoj Ekvivalent Nosti Podmnozhestv Kategorii," Problemi Kibernetiki, Vol. 8 Fizmatgiz, Moskva, 1962, pp. 201–210. Translation: "On Algebraic Equivalence of Subsets of Categories," Problems of Cybernetics, Vol. 8, Joint Publications Research Service, Department of Commerce, Washington, D.C., 1964, pp. 358–376.

16. Glushkov, V. M., "Ob Odnom Metode Automatizatsii Programmirovaniya," Problemi Kibernetki, Vol. 2, Fizmatgiz, 1959, pp. 181–184. Translation: "On a Method of Automatic Programming," Problems of Cybernetics, Vol. 2, Pergamon Press, New York, 1961, pp. 529–533.

17. Iliffe, J. K., "The Use of the Genie System in Numerical Calculation," Annual Review in Automatic Programming, Vol. 2, edited by Richard Goodman, Pergamon Press, New York, 1961, pp. 1-28.

18. Kaluzhnin, L. A., "Ob Algoritmizatsii Matematicheskikh Zadach," Problemi Kibernetiki, Vol. 2, Fizmatgiz, 1959, pp. 51–68. Translation: "On the Algorithmization of Mathematical Problems," Problems of Cybernetics, Vol. 2, 1961, pp. 371–391.

19. Kamynin, S. S., E. Z. Lyubimskii, and M. R. Shura-Bura, "Ob Automatizatsii Programmirovanniya Pri Pomoshchej Programmuruyushchej," Problemi Kibernetiki, Vol. 1, Fizmatgiz, pp. 135-171. Translation: "Automatic Programming with a Programming Program," Problems of Cybernetics, Vol. 1, Pergamon Press, New York, 1960, pp. 149-191.

20. Kitov, A. I., and N. A. Krinitskij, Elektronnye Tsifrovye Mashiny i Programmirovanie (Electronic Digital Computers and Programming), Fizmatgiz, Moskva, 1961.

21. Lyapunov, A. A., "O Logicheskikh Skhemakh Programm," Problemi Kibernetiki, Vol. 1, Fizmatgiz, Moskva, 1958, pp. 46-74. Translation: "The Logical Schemes of Programs," Problems of Cybernetics, Vol. 1, Pergamon Press, Oxford, 1960, pp. 48-81.

22. Lyapunov, A. A., "K Algebraickeskoj Traktovke Programmirovaniya," Problemi Kibernetiki, Vol. 8, Fizmatgiz, Moskva, 1962, pp. 235-242. Translation: "Algebraic Treatment of Programming," Problems of Cybernetics, Vol. 8, Joint Research Publications Service, Department of Commerce, Washington, D.C., 1964, pp. 408-422.

23. Markov, A. A., Theory of Algorithms, Works of the Mathematical Institute V. A. Steklov XLII, published by the Academy of Sciences of USSR, Moscow, 1954. Translation published for the National Science Foundation, Washington, D.C., by the Israel Program for Scientific Translations, 1961.

24. Ter Mikaelyan, T. M., "O Programmakh s Izmenyayushchimsya Poryadkom, Vypolneniya Tsiklov" ("On Programs with a Varying Order of Execution Cycles"), Problemi Kibernetiki Vol. 12, "Nauka," 1964.

25. Minsky, M., "Recursive Unsolvability of Post's Problem of Tag and Other Topics in the Theory of Turing Machines," Annals of Mathematics, Vol. 74, No. 3, 1961, pp. 437-455.

26. Northcott, D. G., An Introduction to Homological Algebra, Cambridge University Press, London, 1960.

27. Podlovchenko, R. I., "Ob Ocnovnykh Ponyatiyakh Programmirovaniya, I," Problemi Kibernetiki, Vol. 1, Fizmatgiz, 1958, pp. 128-134. Translation: "On the Basic Principles of Programming, I," Problems of Cybernetics, Vol. 1, Pergamon Press, Oxford, Paris, New York, 1960, pp. 141-148.

28. Podlovchenko, R. I., "Ob Ocnovykh Ponyatiyakh Programmirovaniya, II," Problemi Kibernetiki, Vol. 3, Fizmatgiz, Moskva, 1960, pp. 123-138. Translation: "The Basic Principles of Programming, II, "Problems of Cybernetics, Vol. 3, Pergamon Press, Oxford, 1962, pp. 885-907.

29. Podlovchenko, R. I., "O Preobrazovaniyakh Skhem Programm i Ikle Premeni v Programmirovanii" ("On the Transformations of Program Schemes and Their Application to Programming"), Problemi Kibernetiki, Vol. 7, Fizmatgiz, Moskva, 1962, pp. 161-188.

30. Rice, H. G., "Classes of Recursively Enumerable Sets and Their Decision Problems, "Transactions of the American Mathematical Society, Vol. 74, No. 2, 1953, pp. 358-366.

31. Rutledge, J. D., "On Yanov's Program Schemata," Journal of the Association for Computing Machinery, Vol. 10, No. 2, 1964, pp. 1-9.

32. Shepherdson, J. C., and H. E. Sturgis, "Computability of Recursive Functions," Journal of the Association for Computing Machinery, Vol. 10, No. 2, 1963, pp. 217-255.

33. Turski, W., "Some Results of Research in Automatic Programming in Eastern Europe," Advances in Computers, edited by F. L. Alt and M. Rubinoff, Academic Press, New York, 1964, pp. 23-108.

34. Wang, H., "A Variant to Turing's Theory of Calculating Machines," Journal of the Association for Computing Machinery, Vol. 4, 1958, pp. 63-92.

35. Yanov, Yu. I., "O Logicheskikh Skhemakh Programm," Problemi Kibernetiki, Vol. 1, Fizmatgiz, Moskva, 1958, pp. 46-74. Translation: "The Logical Schemes of Algorithms," Problems of Cybernetics, Pergamon Press, Oxford, 1960, pp. 82-140.

36. Zaslavskii, I. D., "Graf-Skhemy s Pamyatyii" ("Graph-Schemes with Memory"), Matematycheskogo Instuta Imeni V. A. Steklova, LXXII, pp. 99-192.

DiPaola

A SURVEY OF SOVIET WORK IN THE THEORY OF COMPUTER PROGRAMMING