MEMORANDUM
RM-5136-PR
DECEMBER 1966

SOVIET CYBERNETICS TECHNOLOGY: VIII.
Report on the Algorithmic Language ALGEC
(Final Version)

Translated by Wade B. Holland

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-5136-PR
DECEMBER 1966

SOVIET CYBERNETICS TECHNOLOGY: VIII. Report on the Algorithmic Language ALGEC (Final Version)

Translated by Wade B. Holland

This research is supported by the United States Air Force under Project RAND—Contract No. AF 49(638)-1700—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 and SUMMARY

This Memorandum is a translation of the final, official version of the new Soviet ALGorithmic language for EConomics problems, ALGEC. An earlier, preliminary version of the language description was translated in Part VII in this series of RAND Memoranda;* a summary, analysis, and evaluation of both versions will appear as Part IX (RM-5157-PR).

The development of ALGEC and other general-purpose programming languages† represents a shift away from the traditional Soviet practice of coding in machine language or in special-purpose, machine- or institution-oriented languages. ALGEC in particular reflects the growing concern for an orderly transition to automated, computer-based systems for accounting, national economic control, and centralized planning. As the authors state in their Introduction (see p. 1), ALGEC is an attempt "to better accommodate the specifics of economics problems" and is intended for writing "programs for a number of typical problems involving the processing of economics information..."; ALGEC was designed to facilitate the implementation of translators that will convert programs written in ALGEC to the language of the particular machine being used.

---

*RM-5135-PR; for a listing of all items in the series and a bibliography of RAND publications on Soviet cybernetics and computer technology, see pp. 133-137.

†Ibid., pp. v-vi.

The ALGEC description, authored by a group consisting of M. A. Korolev, K. S. Kuz'min, S. S. Lavrov, A. A. Letichevskii, G. K. Stoliarov, and M. R. Shura-Bura, and under the direction of V. M. Glushkov, appeared in the March-April 1966 issue of the journal Kibernetika, published in Kiev.* ALGEC is based on ALGOL 60,† designed as an international language for handling numerical processing. The Introduction summarizes the intended ALGEC changes to ALGOL 60; an evaluation of the success of the authors in executing their plans is contained in the above-mentioned Memorandum, Part IX in this series, by Dr. Niklaus Wirth of the Computer Science Department at Stanford University.

The translator has added a Russian-English glossary of ALGEC terminology and generated an Index of Definitions of Concepts and Syntactic Units (patterned after the ALGOL 60 index); the Index includes an English-Russian terminology glossary.

This translation will be published in the journal Cybernetics, a cover-to-cover translation of Kibernetika, issued by The Faraday Press, Inc., New York (Vol. 2, No. 2). Reproduction of the translation of the language description for any purpose is explicitly permitted; reference should be made to the translation having been undertaken by The RAND Corporation under U.S. Air Force Project RAND and to the initial open-literature publication in Cybernetics as the source.

---

*"Soobshchenie ob Algoritmicheskom Iazyke ALGEK" ["Report on the Algorithmic Language ALGEC"], Kibernetika [Cybernetics], No. 2, March-April 1966, pp. 57-102.

†See Ref. 1, p. 115.

THE AUTHORS

KOROLEV, Mikhail Antonovich -- Department Head at the Moscow Economic-Statistics Institute, Candidate in the Economic Sciences (the author of the preliminary version of ALGEC).

KUZ'MIN, Kirill Sergeevich -- Laboratory Head at the Central Economic-Mathematics Institute of the Academy of Sciences of the USSR, Moscow.

LAVROV, Sviatoslav Sergeevich -- Professor at Moscow State University, Doctor of Technical Sciences.

LETICHEVSKII, Aleksandr Adol'fovich -- Senior Scientist at the Institute of Cybernetics of the Academy of Sciences of the Ukrainian SSR, Candidate in the Physical-Mathematical Sciences.

STOLIAROV, Gennadii Konstantinovich -- Director of a Section of the Special Design Bureau at the Minsk Radio Factory.

SHURA-BURA, Mikhail Romanovich -- Section Head in the Department of Applied Mathematics at the Steklov Mathematics Institute of the Academy of Sciences of the USSR, Doctor of Physical-Mathematical Sciences.

Although not listed as one of the authors, the design group was under the chairmanship of Academician Viktor Mikhailovich GLUSHKOV, Director of the Institute of Cybernetics of the Academy of Sciences of the Ukrainian SSR. Glushkov is one of the world's leading cyberneticists, and his Kiev Institute of Cybernetics is one of the most prestigious in its field. Glushkov is the Executive Editor of the journal Kibernetika in which the ALGEC report was published; although nominally the official organ of the Ukrainian Academy of Sciences' Department of Mathematics, Mechanics, and Cybernetics, it is usually associated quite closely with the Institute of Cybernetics.

TRANSLATOR'S FOREWORD

Reserved words (underscored) and metalinguistic variables (enclosed in angular brackets, ⟨⟩) have been translated consistently throughout the text. ALGEC can use either the Latin or the Cyrillic alphabet and Russian or English reserved words. A footnote in the original Russian text listed a few of the English equivalents of reserved words; we have used these English equivalents as given (although in some cases we felt the choices were poor). The translations adopted here for reserved words not listed with their English counterparts and for all metalinguistic variables are the personal choices of the translator (although in most cases they are the same as the equivalent ALGOL 60 terms). It is our understanding that an official Russian-English-German terminology dictionary is being planned; if so, it would probably result in many differences in translation from the present version.

We are grateful to Dr. Niklaus Wirth of Stanford University for closely reviewing the translation in the course of preparing his critique of ALGEC. His comments and suggestions resulted in many improvements.

CONTENTS

THE ALGEC REPORT

THE DEVELOPMENT OF ALGEC

(Translation of an Article Entitled, "Report on the Working Sessions of the Group on Algorithmic Languages for Processing Economics Information (GAIAPEI)"*)

[TRANSLATOR'S NOTE: This article, appearing in the issue of Kibernetika immediately preceding that in which the ALGEC Report was published, provides some background to the development of ALGEC and to the concern among countries of the Soviet block for a unified approach to the design and implementation of programming languages and translators. It is interesting to note that the article pays little attention to the antecedents of ALGEC or to the fact that ALGEC's development had been underway for nearly a year at the time of the first GAIAPEI meeting; on the other hand, the ALGEC Report makes no mention of subsequent GAIAPEI sponsorship of ALGEC, other than to say that GAIAPEI has approved publication of the present version.--WH]

GAIAPEI is a working group under the Commission on Multilateral Cooperation [Komissiia mnogostoronnego sotrudnichestva] in the Area of "Scientific Problems of Computer Technology" of the Academies of Sciences of the

---

*"Soobshchenie o Rabochikh Soveshchaniiakh Gruppy Algoritmicheskikh Iazykov po Pererabotke Ekonomicheskoi Informatsii (GAIAPEI)," M. Korolev, Kibernetika (Cybernetics), No. 1, January-February 1966, pp. 101-102; translated by Wade B. Holland.

Socialist Countries. Working in parallel with the Group on Automatic Programming for Intermediate-Size Machines (GAMS), GAIAPEI working sessions have included the participation of delegations from the Academies of Sciences of the People's Republic of Bulgaria, the Hungarian People's Republic, the GDR [German Democratic Republic], the PNR [Polish People's Republic], the Rumanian Socialist Republic, the USSR, and the Czechoslovak Socialist Republic.

The first working session of GAIAPEI took place in Warsaw, October 19-24, 1964. The basic tasks and operational approaches of the group were discussed there; reports by participants on work in designing algorithmic languages for data processing were heard. The feasibilities of using the ALGOL 60 language (proposed by the Soviet delegation) or the COBOL 61 language (proposed by the Polish delegation) as a basis for developing a standard algorithmic data processing language were examined. After hearing all sides, the necessity for development at the present time of both approaches was recognized. The working session requested that the Academies of Sciences of the USSR and the PNR [Poland] prepare plans for the following session for their respective algorithmic languages.

At GAIAPEI's second working session (held in Berlin, March 22-27, 1965), plans for COBOL-GAIAPEI and ALGEC-GAIAPEI languages were discussed and accepted as the basis for further work. To the authors of these proposals was assigned responsibility for the preparation by the next meeting of final plans for their reference languages, taking into consideration conclusions reached during the discussions.

The major focus of the meeting was on questions of unifying concepts and constructs of the languages under consideration. The participants heard and accepted a plan for a GAIAPEI alphabet for the working group, prepared by a commission created at the meeting. This document specifies both the composition of the alphabet and the ordering of its symbols. GOST [State Standard] 10859-64, "Machines, computing. Alphanumeric codes for punchcards and paper tape" (USSR), was used as a basis in developing the reference alphabet. A session decision stipulated that all working materials of GAIAPEI must use the reference alphabet exclusively, and that the descriptions of the hardware representations of COBOL-GAIAPEI and ALGEC-GAIAPEI be accompanied by rules for establishing uniformity between the symbols of their languages and the reference alphabet symbols and by provisions for the adopted alphabetical ordering.*

The delegation of the GDR [German] Academy of Sciences was assigned the preparation for the next meeting of a plan for terminological, syntactic, and semantic unification of the basic concepts and constructs of the ALGEC-GAIAPEI and the COBOL-GAIAPEI languages. Between the second and third working sessions of GAIAPEI, representatives of the GDR delegation met with representatives of the Polish delegation (in Warsaw, June 24-27, 1965) and the Soviet delegation (in Moscow, June 27-July 5, 1965) and discussed with them problems connected with defining

---

*See fn., pp. 8, 42.--Trans.

some of the concepts of the languages under examination. Agreement was reached on a number of basic points, and a plan was prepared for a terminology dictionary in Russian, English, and German.

Representatives of the Institute of Cybernetics of the Academy of Sciences of the Ukrainian SSR and of the Central Economic-Mathematics Institute of the Academy of Sciences of the USSR, in addition to the members of the delegations, took part in the work of the third working session (in Tashkent, October 24-31, 1965). The plans for the ALGEC-GAIAPEI and the COBOL-GAIAPEI languages were discussed, and were accepted and forwarded to the Commission on Multilateral Cooperation in the Area of "Scientific Problems of Computer Technology" of the Academies of Sciences of the Socialist Countries, with a request that these languages be recommended for algorithmic description of economics problems and that translators be devised for their use in the cooperating countries. It was recommended that the COBOL-GAIAPEI language be published in early 1966 in the [Polish] journal Algoritmy, and the ALGEC-GAIAPEI language in one of the publications of the Academy of Sciences of the USSR. Comments from the meeting relating to, among other things, a unified description of the formats for both languages must be considered in preparing them for publication.

A plan for the development in the cooperating countries during the 1965-68 period of translators for both languages for small, intermediate, and large computers was recommended to the Commission on Multilateral

Cooperation of the Academies of Sciences of the Socialist Countries. The working session recommended also that the Commission take charge of the material developed by the session on specifying requirements for the COBOL-GAIAPEI and ALGEC-GAIAPEI translators.

The description of the translators must be done with the help of a special symbolic language, a plan for which is being prepared by the Polish delegation and will be discussed at the next meeting.

Since ALGEC-GAIAPEI was accepted by the session without input/output procedures, the development of these procedures must be completed by the Academy of Sciences of the USSR by the next meeting. The approach to this work was specified at the session's third meeting.

The session approved material on a terminology dictionary for COBOL-GAIAPEI, prepared by the GDR deletation, and recommended its acceptance as the basis for further work on a joint GAIAPEI terminology dictionary.

A preliminary plan for the DAIA-2 language (a subset of ALGEC-GAIAPEI), prepared by the delegation of the Czechoslovak SSR, was also considered at the working session.

The next (fourth) working session of GAIAPEI must be convened in April 1966 in Budapest.

--M. Korolev

INTRODUCTION

A design group charged with the development of an algorithmic language for describing economics problems (including problems in planning, accounting, and statistics) was created in November 1963 under the chairmanship of Academician V. M. Glushkov, in accordance with a resolution of a meeting called by the Main Administration on the Introduction of Computer Technology under the USSR State Committee on the Coordination of Scientific Research Work.

At its sessions, the design group resolved to base the language on the ALGOL 60 international algorithmic language, eliminating some parts of it that were lacking in precise interpretation or which created significant difficulty in designing effective translators. In order to better accommodate the specifics of economics problems, it was decided to introduce into the language a number of additional features permitting:

1) description of documents and sets of documents of comparatively complex structure with large and usually variable quantities of data (various tables, records, orders, indices, etc.);

2) description of means for selecting and processing different items of information contained in such documents;

3) facility for handling textual information with access to any textual element.

The present report is a description of such a language, called ALGEC (ALGorithmic language for EConomics problems). Wide use has been made throughout the report of material on

the ALGOL 60 language,[1] its derivative SUBSET ALGOL 60 (IFIP),[2] and the input/output procedures developed for ALGOL.[3] Sections wholly or partially derived from these sources (i.e., all sections except 2.7, 3.4, 3.7, 3.8, 3.9, and 5.3) constitute the larger part of this report; specific references are not given in the text. In addition, ideas from the COBOL-61 language[4] and its supplement on input/output procedures[5] have been used in this work.

The proposed language can be used to write programs for a number of typical problems involving the processing of economics information and is in a form suitable for the design of translators. The authors are aware that in the course of this work shortcomings in the language will become apparent and may necessitate some refining of the language.

The first versions of the language were considered at several meetings and seminars. The projected language was distributed among a number of organizations. Their comments helped the authors in selecting the final version. The authors convey their appreciation to all the individuals and organizations who submitted reviews and to those who participated in the discussions.

Great assistance to the work of the designers was rendered by Comrades Iu. Ia. Basilevskii, M. N. Yefimova, and A. S. Frolov, to whom the authors express gratitude.

The present publication has been approved by the Group on Algorithmic Languages for Processing Economics Information (GAIAPEI) under the Commission of the Academies of Sciences of the Socialist Countries on Multilateral

Cooperation in the Area of the Scientific Problems of Computer Technology and is recommended at the present time in the cooperating countries for describing economics problems and for the design of translators.

GAIAPEI recommends that the authors of the language proceed to the work of creating an input/output apparatus and delegates to them authority to introduce into the language corrections which become apparent in connection with designing the indicated apparatus, with developing translators for the language, and with an accumulation of experience in handling the algorithmic description of economics problems. All indicated corrections to the language must be approved by GAIAPEI.

1. STRUCTURE OF THE LANGUAGE

The ALGEC algorithmic language has two levels: a reference language, and various hardware representations. The reference language is the working language for all ALGEC users, it serves as the model for all hardware representations, and is also the basis and the guide for designing translators. As far as possible, it has been adapted to the publication of algorithms, inasmuch as its symbols are defined so as to facilitate mutual understanding and are not machine-limited or derived from some programmer or pure mathematical notation. Publications on ALGEC must use only the reference representation.

Each hardware representation is thus a condensation of the reference language as dictated by the limited number of characters on the particular computer's input equipment. Each uses the existing set of characters and is the input language of the translator for that computer. Each hardware representation must be accompanied by a special set of rules for translation from the reference language and back.

The present description is given in terms of the reference representation. This means that all objects defined within the language are represented by a specified set of so-called basic symbols. Any hardware representation can differ from the reference representation only in its choice of symbols. Structure and content must be the same for all representations.

The purpose of the ALGEC algorithmic language is to describe the processing of data characteristic to economics

problems. In this regard, in addition to a fully-developed system for describing wholly computational processes, means are included in the language for describing, retrieving, and transferring data organized in arbitrary hierarchical structures consisting of string values (sequences of symbols), in addition to numerical and logical values.

The basic concept used to describe the data processing is that of an expression containing as constituent parts constants, variables, and functions, joined together by operator signs. The values of expressions can be assigned to other quantities by means of explicit formulae called assignment statements.

To show the flow of the data processing, certain other statements have been added, in addition to statement selection conditions; these latter can describe, for example, alternatives for or iterative repetitions of statements. In order that these statements can function, there arises the necessity for referencing statements; in this regard, statements can be provided with labels. To join a sequence of statements into one compound statement, it can be enclosed in the statement brackets begin and end.

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements. These properties can be, for example, the class of permitted values of variables, the dimension of an array of numbers, the hierarchical structure of quantities, or even the set of rules defining some function. A sequence of declarations followed by a sequence of statements and enclosed between begin and end constitute

a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement not contained within another statement and which makes no use of other statements not contained within it.

The syntactic rules and the semantics of the language are given below.*

1.1 FORMALISM FOR SYNTACTIC DESCRIPTION

Syntax is described with the aid of metalinguistic formulae.[6] Their interpretation is best explained by an example:

⟨ab⟩ ::= (|[|⟨ab⟩(|⟨ab⟩⟨d⟩

Sequences of characters enclosed in the brackets ⟨⟩ represent metalinguistic variables whose values are sequences of symbols. The marks ::= (with the meaning "is defined as") and | (with the meaning "or") are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus, the formula above gives a recursive rule

---

*Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed and the kind of arithmetic assumed, as well as the course of action to be taken in all such cases as may occur during the execution of the computation.

for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ can have the value (or [, or that given some permissible value of ⟨ab⟩, another value can be formed by following it with the character (or with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

```
[(((1(37(
     (122345(
     (((
     [86
```

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e., the sequences of characters appearing within the brackets ⟨⟩, such as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words that have appeared in this manner are used elsewhere in the text, they always refer to the corresponding syntactic definition. In addition, some formulae are given several times.

Definition:

⟨empty⟩ ::=

(i.e., the null string of symbols).

2. BASIC SYMBOLS, IDENTIFIERS, NUMBERS, AND STRINGS. BASIC CONCEPTS

The reference language is built up from the following basic symbols:*

⟨basic symbol⟩ ::= ⟨quotation symbol⟩ | ⟨quotation mark⟩

⟨quotation symbol⟩ ::= ⟨letter⟩ | ⟨digit⟩ | ⟨logical value⟩ | ⟨special character⟩ | ⟨delimiter⟩

2.1 LETTERS

⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V| W|X|Y|Z|Б|Г|Д|Ж|З|И|Й|Л|П|Ф|Ц|Ч|Ш|Щ|Ы|Ь|Э|Ю|Я|Ъ

This alphabet in a hardware representation can be restricted, or can be extended by the addition of any other distinctive symbols (i.e., symbols not coinciding with any digit, logical value, special character, delimiter, or quotation mark).

Letters do not have individual meaning. They are used for forming identifiers and quotations† (see sections 2.4 and 2.6).

---

*Based on the alphabet adopted in GOST [USSR State Standard] 10859-64. Machines, computing. Alphanumeric codes for punchcards and papertape.

The ALGEC alphabet is a subset of the GAIAPEI reference alphabet, which includes the capital letters of the Russian alphabet, the lower-case letters of the Latin alphabet, and all the remaining symbols of GOST 10859-64.

†It should be particularly noted that throughout the reference language underlining is used for defining independent basic symbols (see sections 2.2.2 and 2.3). It is understood that these symbols have no relation to the individual letters of which they are composed. Within the present report underlining will be used for no other purpose [except in section headings].

2.2 DIGITS, LOGICAL VALUES, SPECIAL CHARACTERS, AND QUOTATION MARKS

2.2.1 Digits

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers, and quotations.

2.2.2 Logical Values

⟨logical value⟩ ::= true | false

The logical values have a fixed obvious meaning.

2.2.3 Special Characters

⟨special character⟩ ::= ¯|%|◇|↑|_|–|⁝|≫|°|´|→|?|↓|∅|±|▽

The special characters are used only in quotations.

2.2.4 Quotation Marks

⟨quotation mark⟩ ::= ‘|’

Quotation marks are used only for delimiting quotations.

2.3 DELIMITERS

⟨delimiter⟩ ::= ⟨operator⟩ | ⟨separator⟩ | ⟨bracket⟩ | ⟨declarator⟩

⟨operator⟩ ::= ⟨arithmetic operator⟩ | ⟨relational operator⟩ | ⟨logical operator | ⟨string operator⟩ | ⟨sequential operator⟩

⟨arithmetic operator⟩ ::= +|-|×|/|÷|↑

⟨relational operator⟩ ::= <|≤|=|≥|>|≠

⟨logical operator⟩ ::= ≡|⊃|∨|∧|¬

⟨string operator⟩ ::= text | sense | quotation | ←

⟨sequential operator| ::= to | if | then | else | for | do

⟨separator⟩ ::= ,|.|$_{10}$|:|;|␣|*| step | until | while |
comment | element

⟨bracket⟩ ::= (|)|[|]| begin | end

⟨declarator⟩ ::= Boolean | real | integer | string |
format | array | compound | switch | procedure | as |
label | value

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place below.*

Such typographical features as a blank space or change to a new line have no significance in the reference language. They may, however, be used to facilitate reading.

For the purpose of including text among the symbols of a program, the following rules for comments hold:

The sequence of basic symbols:	is equivalent to
; comment ⟨any sequence not containing ; ⟩ ;	;
begin comment ⟨any sequence not containing ; ⟩ ;	begin
end ⟨any sequence containing neither end nor ; nor else⟩	end

*For the underlined words representing the basic symbols, changes in gender and number are permissible. For example, логическое, логический, логическая and логические [Boolean in neuter, masculine, feminine, and plural nominative forms, respectively] represent the same basic reserved word. In actual representations, the use of the equivalent English words is also permitted....

By equivalence is here meant that any of the three structures shown in the left-hand column can, in any occurrence outside a quotation, be replaced by the symbol shown in the right-hand column; this replacement has no effect on the action of the program. It is further understood that the comment structure encountered first in a text when reading from left to right must take precedence in being replaced over later structures contained in that sequence.

Symbols other than the basic symbol reserved words can be used in the texts of comments.

2.4 IDENTIFIERS

2.4.1 Syntax

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ |
⟨identifier⟩⟨digit⟩

2.4.2 Examples

A
A12
A2
TIME CARD
PAYROLL ACCOUNT FOR THE PERIOD FROM 26 THROUGH 31

2.4.3 Semantics

Identifiers have no inherent meaning, but serve to denote simple variables, arrays, compounds, labels, switches, and procedures. They can be chosen arbitrarily (see, however, sections 3.2.4 and 4.7.7).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declaration of the program (see section 2.8) or are elements of a list structure (see section 5.3.5).

2.5 NUMBERS

2.5.1 Syntax

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩

⟨integer⟩ ::= ⟨unsigned integer⟩ | + ⟨unsigned integer⟩ |
- ⟨unsigned integer⟩

⟨proper fraction⟩ ::= . ⟨unsigned integer⟩

⟨exponent⟩ ::= $_{10}$ ⟨integer⟩

⟨decimal number⟩ ::= ⟨proper fraction⟩ | ⟨unsigned integer⟩
⟨proper fraction⟩

⟨unsigned number⟩ ::= ⟨unsigned integer⟩ | ⟨decimal number⟩ |
⟨exponent⟩ | ⟨decimal number⟩⟨exponent⟩

⟨number⟩ ::= ⟨unsigned number⟩ | + ⟨unsigned number⟩ |
- ⟨unsigned number⟩

2.5.2 Examples

0	-200.084	$-.083_{10}-02$
177	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$2_{10}-4$	$+_{10}+5$

2.5.3 Semantics

Decimal numbers have their conventional meaning. The exponent is a scale factor expressed as an integral power of 10.

2.5.4 Types

Integers are of type integer. All other numbers are of type real (see section 5.1, Type Declarations).

2.6 QUOTATIONS

2.6.1 Syntax

⟨blank quotation⟩ ::= ⟨quotation symbol⟩ | ⟨blank quotation⟩ ⟨quotation symbol⟩ | ⟨empty⟩

⟨quotation element⟩ ::= ⟨quotation symbol⟩ | ⟨quotation⟩

⟨open quotation⟩ ::= ⟨quotation element⟩ | ⟨open quotation⟩ ⟨quotation element⟩ | ⟨empty⟩

⟨quotation⟩ ::= ‘ ⟨open quotation⟩ ’

2.6.2 Examples

‘5K,,–ⁱ“[[[ⁱ“^ =|:’TV’’

‘..THIS␣IS␣A␣‘QUOTATION’’

‘LOAD␣‘...’ORGANIZATION‘...’AT␣ADDRESS‘...’’

2.6.3 Semantics

In order to provide facility to handle an arbitrary sequence of quotation symbols as a single integer, the quotation marks ‘ and ’ have been introduced. The symbol ␣ denotes a one-character space (blank). It has no

significance outside quotations. The two ' ' symbols, one following directly after the other, denote an empty quotation.

Open quotations can be used as values of variables having a string type declaration, as values of string expressions, and as values of functions (see section 3.4, String Expressions). The value of a string expression, represented by a quotation, is obtained by eliminating the pair of outer quotes. Each open quotation element occupies a separate position. The positions are considered as being numbered from left to right.

2.7 FORMATS

2.7.1 Syntax

⟨repeat⟩ ::= (⟨subscript expression⟩)
⟨space⟩ ::= ␣|␣ ⟨repeat⟩
⟨P positions⟩ ::= P | P ⟨repeat⟩
⟨P part⟩ ::= ⟨P positions⟩ | ⟨P part⟩⟨P positions⟩ |
⟨P part⟩⟨space⟩
⟨* positions⟩ ::= * | * ⟨repeat⟩
⟨* part⟩ ::= ⟨* positions⟩ | ⟨* part⟩⟨* positions⟩ |
⟨* part⟩⟨space⟩
⟨suppressed part⟩ ::= ⟨P part⟩ | ⟨* part⟩
⟨9 positions⟩ ::= 9 | 9 ⟨repeat⟩
⟨9 part⟩ ::= ⟨9 positions⟩ | ⟨9 part⟩⟨9 positions⟩ |
⟨9 part⟩⟨space⟩
⟨integer part⟩ ::= ⟨suppressed part⟩ | ⟨9 part⟩ |
⟨suppressed part⟩⟨9 part⟩
⟨space series⟩ ::= ⟨empty⟩ | ⟨space series⟩⟨space⟩
⟨unsigned integer format⟩ ::= ⟨space series⟩⟨integer part⟩

```
⟨M series⟩ ::= M | ⟨M series⟩ M
⟨scale⟩ ::= ⟨M series⟩⟨space series⟩ | M ⟨repeat⟩⟨space series⟩
⟨+ positions⟩ ::= + | + ⟨repeat⟩
⟨+ part⟩ ::= ⟨+ positions⟩ | ⟨+ part⟩⟨space series⟩⟨+ positions⟩
⟨- positions⟩ ::= - | - ⟨repeat⟩
⟨- part⟩ ::= ⟨- positions⟩ | ⟨- part⟩⟨space series⟩⟨- positions⟩
⟨sign part⟩ ::= ⟨empty⟩ | ⟨space series⟩⟨+ part⟩ |
     ⟨space series⟩⟨- part⟩
⟨integer format⟩ ::= ⟨unsigned integer format⟩
     + ⟨space series⟩ | ⟨unsigned integer format⟩
     - ⟨space series⟩ | ⟨sign part⟩⟨unsigned integer format⟩
⟨point designator⟩ ::= . ⟨space series⟩ | T ⟨space series⟩
⟨roundoff designator⟩ ::= ⟨empty⟩ | U ⟨space series⟩
⟨proper fraction format⟩ ::= ⟨point designator⟩⟨9 part⟩
     ⟨roundoff designator⟩
⟨proper fraction scaled format⟩ ::= ⟨scale⟩⟨9 part⟩
     ⟨roundoff designator⟩
⟨scale designator⟩ ::= ⟨roundoff designator⟩ | ⟨scale⟩
     ⟨roundoff designator⟩
⟨decimal number format⟩ ::= ⟨unsigned integer format⟩
     ⟨scale designator⟩ | ⟨space series⟩⟨proper fraction format⟩ |
     ⟨unsigned integer format⟩⟨proper fraction format⟩ |
     ⟨space series⟩⟨proper fraction scaled format⟩
⟨mantissa format⟩ ::= ⟨space series⟩⟨9 part⟩ |
     ⟨space series⟩⟨9 part⟩⟨roundoff designator⟩ | ⟨space series⟩
     ⟨proper fraction format⟩⟨space series⟩⟨9 part⟩
     ⟨proper fraction format⟩
⟨mantissa sign⟩ ::= ⟨space series⟩ + | ⟨space series⟩ -
⟨signed mantissa format⟩ ::= ⟨mantissa format⟩ |
     ⟨mantissa sign⟩⟨mantissa format⟩
```

⟨exponent format⟩ ::= + ⟨space series⟩⟨9 part⟩ |
- ⟨space series⟩⟨9 part⟩
⟨real format⟩ ::= ⟨sign part⟩⟨decimal number format⟩ |
⟨decimal number format⟩ + ⟨space series⟩ |
⟨decimal number format⟩ - ⟨space series⟩ |
⟨signed mantissa format⟩ $_{10}$ ⟨space series⟩⟨exponent format⟩
⟨number format⟩ ::= ⟨integer format⟩ | ⟨real format⟩
⟨insert⟩ ::= ⟨space⟩ | ⟨quotation⟩
⟨S positions⟩ ::= S | S ⟨repeat⟩
⟨E positions⟩ ::= E | E ⟨repeat⟩
⟨string format⟩ ::= ⟨S positions⟩ | ⟨E positions⟩
⟨insert⟩⟨string format⟩ | ⟨string format⟩⟨insert⟩ |
⟨string format⟩⟨S positions⟩ | ⟨string format⟩
⟨E positions⟩
⟨format⟩ ::= ⟨number format⟩ | ⟨string format⟩

2.7.2 Examples

PPPP	␣␣99␣999␣␣
P(4)	␣␣**␣***␣␣
**	␣999M(3)␣
*9	␣+M(5)␣9(6)U␣␣
PP9	+(6)9.9(3)U
9(6)	␣(4)-(6)9T9(3)U␣(3)
9(N+M)	␣(4)-9.9(5)$_{10}$-99␣
+++99	ES(3)␣␣E(2)
--PP9	S(P×Q)
**9-	␣SSS␣'DOL'␣SS␣'CTS'␣
*9U	␣SS'-TH JANUARY'

2.7.3 Semantics

A format imposes certain restrictions on the structure of the string values with which it is associated (see sections 3.4, 5.1, and 5.2).

Formats are open quotations (see section 2.6, Quotations) representing permissible values of format expressions.

A subscript expression (see section 3.1.4.2) contained in a repeat must have a positive value; where this value is n, the symbol preceding the repeat is repeated n times; e.g., ␣(4) in a format (outside an insert quotation) is equivalent to ␣␣␣␣. This means that any construction of the form S ⟨repeat⟩, where S is a symbol that, according to the syntax, can precede a repeat, is the same as the sequence

$$\underbrace{S.....S}_{n \text{ times}} ,$$

where n is the value of the subscript expression in the repeat.

2.8 QUANTITIES, KINDS AND SCOPES

The following kinds of quantities are distinguished: simple variables, arrays, compounds, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions within which the declaration of the first-level identifier associated with that quantity is valid (see section 5, Declarations). The scope of a label is determined in accordance with section 4.1.3.

2.9 VALUES AND TYPES

A value is some ordered set, the elements of which can be numbers, open quotations, and logical values (special cases: an ordered set of numbers or an individual number, an ordered set of open quotations or an individual open quotation, an ordered set of logical values or a single logical value), or some label.

Certain of the syntactic units are said to possess values. In general, these values change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of a subarray designator and an array identifier is the ordered set of values of the corresponding array of subscripted variables (see section 3.1.4.1), while the value of a constituent designator is the ordered set of values of its corresponding variable-constituents (see section 3.1.5).

The various types (integer, real, Boolean, string) basically denote properties of values. Values declared as compounds can consist of components with differing value properties. The types associated with syntactic units refer to the values of these units.

3. EXPRESSIONS

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, string, Boolean, designational, and multicomponent expressions. The constituents of these expressions are numbers, quotations, logical values, variables, function designators, subarray indicators, and constituent indicators; arithmetic, logical, and string operators; relational operators, sequential operators, and parentheses. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and of their constituents, is necessarily recursive.

⟨expression⟩ ::= ⟨arithmetic expression⟩ | ⟨string expression⟩ |
⟨Boolean expression⟩ | ⟨designational expression⟩ |
⟨multicomponent expression⟩

3.1 VARIABLES

3.1.1 Syntax

⟨variable identifier⟩ ::= ⟨identifier⟩
⟨array identifier⟩ ::= ⟨identifier⟩
⟨simple compound identifier⟩ ::= ⟨identifier⟩
⟨compound-array identifier⟩ ::= ⟨identifier⟩
⟨subscript expression⟩ ::= ⟨arithmetic expression⟩
⟨position list element⟩ ::= ⟨subscript expression⟩ |
⟨subscript expression⟩ : ⟨subscript expression⟩

⟨position list⟩ ::= ⟨position list element⟩ |
⟨position list⟩ , ⟨position list element⟩
⟨simple variable⟩ ::= ⟨variable identifier⟩ |
⟨variable identifier⟩ [element ⟨position list⟩]
⟨subscript list⟩ ::= ⟨subscript expression⟩ |
⟨subscript list⟩ , ⟨subscript expression⟩
⟨subscripted variable⟩ ::= ⟨array identifier⟩ [⟨subscript list⟩] |
⟨array identifier⟩ [⟨subscript list⟩ element ⟨position list⟩]
⟨variable-constituent tail⟩ ::= ⟨variable identifier⟩ |
⟨variable identifier⟩ [element ⟨position list⟩] |
⟨array identifier⟩ [⟨subscript list⟩] |
⟨array identifier⟩ [⟨subscript list⟩ element ⟨position list⟩] |
⟨simple compound identifier⟩ . ⟨variable-constituent tail⟩ |
⟨compound-array identifier⟩ [⟨subscript list⟩] .
⟨variable-constituent tail⟩
⟨variable-constituent⟩ ::= ⟨variable-constituent tail⟩
⟨variable⟩ ::= ⟨simple variable⟩ | ⟨subscripted variable⟩ |
⟨variable-constituent⟩

3.1.2 Examples

SHOP
TABLE NUMBER [element I]
PART [elements I : K+1]
AI[2, E[6, B, 3]]
RATE [T, S × N/M, D elements 1, 3, 4, 6 : 9]
A[I]. B[K]
A2. R. L
TABLE [NUMBER]. SECTOR [S1, S2]. ZONE [P elements 4 : 8]

3.1.3 Semantics

A variable is a designation given to a single value. This value can be used in expressions for forming other values and can be changed at will by means of assignment statements (see section 4.2).

The type of the value of a given variable is defined by the declaration for the variable itself (see section 5.1, Type Declarations) or for the corresponding array identifier (see section 5.2, Array Declarations); these declarations can be elements of the declaration of a list structure (see section 5.3, Compound Declarations).

3.1.4 Subscripts

3.1.4.1 Subscripted variables designate values which are components of multidimensional arrays (see section 5.2, Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical values of its subscripts (see section 3.3, Arithmetic Expressions).

3.1.4.2 Each subscript position acts like a variable of type integer and the evaluation of the subscript expression on the position list is understood to be equivalent to assigning the value to this fictitious variable (see section 4.2.7.2). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (see section 5.2, Array Declarations).

3.1.4.3 A position list can be included only in a string type variable or in a variable with the proper format as specified by its declaration (see sections 5.1 and 5.2). The individual positions of the elements of the string value (see section 5.1.3.2) of the corresponding variable without a position list are enumerated in this list. The sequence of these elements makes up the value of the given variable. If the position list consists of only one subscript expression, and the corresponding element is a quotation, then an open quotation obtained from this quotation by removing its pair of outer quotation marks serves as the value of the variable.

A colon in a position list element denotes a sequential enumeration of positions, starting from the subscript expression immediately to its left and finishing with the subscript expression at its right. If the value of the second expression is less than the value of the first expression, then that element of the position list distinguishes the null set positions.

As an example, for the string variable A, the value of which is defined by the expression, "ЯЗЫК␣'АЛГЭК'', conversion to specified position values is handled in the following manner:

Variable	Value
A[element 1]	Я
A[elements 3 : 2, 4]	К
A[element 6]	АЛГЭК
A[elements 1 : 1, 2, 6]	ЯЗ 'АЛГЭК'
A[elements 1 : 6]	ЯЗЫК 'АЛГЭК'

The value of variable A[element 7] is not defined for the given example.

A position list must provide the sequence of elements in the strictly ascending order of their position numbers.

3.1.5 Variable-Constituents

A variable-constituent serves to name an individual value that is a primary component of a list (see section 5.3, Compound Declarations). Simple compound identifiers and the names of compound-array components (in the form of subscripted variables in which the given primary component is included) are sequentially enumerated in this name for all levels beginning with the first. The last level indicates the identifier of the variable itself or, possibly, an individual array component with the indicated list of element positions. The names of the different levels are separated by periods.

A variable-constituent of the form

N. I1. I2.....IK. VCT

(where N is some sequence of names; I1, I2,...,IK are simple compound identifiers; and VCT is a variable-constituent tail) can be abbreviated as

N. VCT .

The abbreviated notation is equivalent to the complete notation if there is no possibility of ambiguity arising as a result of the abbreviation, and if and only if a variable-constituent notation of the form

N. VCTI

(where N is the same sequence of names, and VCTI is a variable-constituent tail with the same initial identifier as VCT) can arise only when the full notation of the variable-constituent N. I1. I2.....IK. VCTI is abbreviated.

3.2 FUNCTION DESIGNATORS

3.2.1 Syntax

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨actual parameter⟩ ::= ⟨expression⟩ | ⟨array identifier⟩ |
⟨procedure identifier⟩ | ⟨switch identifier⟩

⟨letter quotation⟩ ::= ⟨letter⟩ | ⟨letter quotation⟩⟨letter⟩

⟨parameter delimiter⟩ ::= , |) ⟨letter quotation⟩ : (

⟨actual parameter list⟩ ::= ⟨actual parameter⟩ |
⟨actual parameter list⟩⟨parameter delimiter⟩
⟨actual parameter⟩

⟨actual parameter part⟩ ::= (⟨actual parameter list⟩) |
⟨empty⟩

⟨function designator⟩ ::= ⟨procedure identifier⟩
⟨actual parameter part⟩

3.2.2 Examples

```
LENGTH (ADDRESS)
MULTIPLY ('␣—X—', 20)
CALCULATE LAST INTERVALS (PART CODE [I])
LIBRARY ('SHIFT', PART CODE [I]) DIRECTION:
  ('RIGHT') SIZE: (CALCULATE LAST INTERVALS)
AVERAGE (PROCESS, N)
```

3.2.3 Semantics

Function designators define single numerical, logical, or string values, obtained as the result of the application of given sets of rules defined by a procedure declaration (see section 5.5, Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Not every procedure declaration defines the value of a function designator.

3.2.4 Standard Functions

Certain identifiers should be reserved for the standard functions of analysis, which are expressed as procedures. It is recommended that this reserved list should contain:

ABS(E)	for the modulus (absolute value) of the value of the expression E
SIGN(E)	for the sign of the value of E (+1 for E > 0, 0 for E = 0, -1 for E < 0)
ENTIER(E)	for the largest integer not greater than the value of E
SQRT(E)	for the square root of the value of E
SIN(E)	for the sine of the value of E
COS(E)	for the cosine of the value of E
ARCT(E)	for the principal value of the arctangent of the value of E
LN(E)	for the natural logarithm of the value of E
EXP(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield values of type real, except for SIGN(E) and ENTIER(E) which have values of type integer. In particular representations these functions can be used without explicit declarations (see section 5, Declarations).

Additionally, it is recommended that the reserved list include the functions LENGTH(T) and SIZE(T), defined for string expressions and variables, and also for variables of type real or integer if in the appropriate declaration a format expression is given. The value of the function designator LENGTH(T) is equal to the number of basic symbols in the current string value of the argument. The value of the function designator SIZE(T) is equal to the number of quotation elements in the current string value of the argument. If the string value of expression T is a blank quotation, then the values of both function indicators are the same.

The effect of a function is clear from the following examples:

T	Length(T)	Size(T)
''	0	0
'␣␣␣'	3	3
'␣␣5␣DECEMBER␣1964'	17	17
'+␣␣01.99$_{10}$-2'	11	11
'A'B''	4	2
M←N	LENGTH(M) + LENGTH(N)	SIZE(M) + SIZE(N)

Accessing a LIBRARY standard procedure (see section 4.7.7) is permitted as a use for a function designator only if the appropriate library subroutine specifies a separate value as the result of its operation.

3.2.5 Transfer Functions

The existence of functions for transferring values of functions of one type into appropriate values of another type is understood. The operation of these functions is described in section 4.2.7.

3.3 ARITHMETIC EXPRESSIONS

3.3.1 Syntax

⟨adding operator⟩ ::= + | -

⟨multiplying operator⟩ ::= × | / | ÷

⟨primary arithmetic expression⟩ ::= ⟨unsigned number⟩ |
⟨variable⟩ | ⟨function designator⟩ |
⟨secondary string expression⟩ sense ⟨format expression⟩ |
(⟨arithmetic expression⟩)

⟨factor⟩ ::= ⟨primary arithmetic expression⟩ |
⟨factor⟩ ↑ ⟨primary arithmetic expression⟩

⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩⟨multiplying operator⟩⟨factor⟩

⟨simple arithmetic expression⟩ ::= ⟨term⟩ | ⟨adding operator⟩
⟨term⟩ | ⟨simple arithmetic expression⟩⟨adding operator⟩
⟨term⟩

⟨if clause⟩ ::= if ⟨Boolean expression⟩ then

⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩ |
⟨if clause⟩⟨arithmetic expression⟩ else
⟨arithmetic expression⟩

3.3.2 Examples

```
25.6
7.394₁₀-8
WAGE
TARIFF TABLE [POSITION]. RATE
MAX (PROCESS, N)
(BALANCE + INCOME - EXPENDITURE)
T sense '+99.99'
X↑2 + Y↑2
if DEDUCTION then if D > 100 then 0.97 else 1 else if
     D ≤ 50 then 1.05 else 1.025
if TAX SUM ≤ 60 then 0 else if TAX SUM ≤ 70 then 0.41
     × (TAX SUM - 60) else 0.06 × TAX SUM
```

3.3.3 Semantics

An arithmetic expression is a rule for computing a numerical value. In the case of simple arithmetic expressions, this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primary arithmetic expressions included in the given expression, as explained in detail in section 3.3.4 below.

The actual numerical value of a primary arithmetic expression is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (see section 5.4.4, Values of Function Designators). These rules are applied to the current values of the procedure parameters given in the expression.

The sense operator in a primary arithmetic expression is defined if and only if the value of its right-hand

operand is numerical in format (see section 2.7) and the value of its left-hand operand is the notation of a number appropriate to that format (see section 3.4.6). The result of the operation is the numerical value defined by that notation. More precisely, the value of the left-hand operand must be a feasible value for a string expression of the type

⟨primary arithmetic expression⟩ text ⟨format expression⟩

with the same right-hand operand value as in the original sense operator. The sense operator is defined such that the value of the expression

PSE sense FE1 text FE2 ,

wherein format expressions FE1 and FE2 have the same value, matches the value of the PSE secondary string expression.

Finally, the values of arithmetic expressions enclosed in parentheses are obtained recursively from the values of the other three kinds of primary expressions.

The value of an arithmetic expression of the form

if BE then AE1 else AE2 ,

where BE is a Boolean expression (see section 3.5, Boolean Expression), and AE1 and AE2 are arithmetic expressions, is recursively defined. If the value of BE is true, the value of the entire arithmetic expression is the value of arithmetic expression AE1; if the value of BE is false, the value of the entire arithmetic expression matches the value of arithmetic expression AE2. According to the syntax, the symbols then and else in an arithmetic expression are used analogously to opening and closing brackets.

3.3.4 Operators and Types

The constituents of simple arithmetic expressions (excluding Boolean expressions used in if clauses and operands of the sense operator) must be of types real or integer (see section 5.1, Type Declarations). The sense of the basic operators and the types of their results are given by the following rules:

3.3.4.1 The operators +, -, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the result is integer if both operands are of type integer, otherwise real.

3.3.4.2 The / and ÷ operators denote division, carried out with due regard for the rules of precedence (see section 3.3.5). Thus, for example,

A/B × 7/(R - S) × D/E

means

((((A/B) × 7)/(R - S)) × D)/E .

The operator / is defined for all four combinations of types real and integer and will yield results of type real in any case.

The operator ÷ is defined only for two operands, both of type integer. The result is also of type integer, mathematically defined as follows:

A ÷ B = SIGN(A/B) × ENTIER(ABS(A/B)) .

3.3.4.2 The operator ↑ in the construction ⟨factor⟩ ↑ ⟨primary arithmetic expression⟩ denotes exponentiation, where the factor is the base and the primary arithmetic expression is the exponent. Thus, for example,

$2 \uparrow N \uparrow K$ means $(2^N)^K$,

while

$2 \uparrow (N \uparrow M)$ means $2^{(N^M)}$.

Writing I for a number of type integer, R for a number of type real, and A for a number of either integer or real type, the result is given by the following rules:

A ↑ I	If I > 0, then A × A × ... × A (I times), of the same type as A.
	If I = 0 and A ≠ 0, then 1, of the same type as A.
	If I = 0 and A = 0, then undefined.
	If I < 0 and A ≠ 0, then 1/(A × A × ... × A) (the denominator has (-I) factors), of type real.
	If I < 0 and A = 0, then undefined.
A ↑ R	If A > 0, then EXP(R × LN(A)), of type real.
	If A = 0 and R > 0, then 0.0, of type real.
	If A = 0 and R ≤ 0, then undefined.
	If A < 0, then always undefined.

3.3.5 Precedence of Operators

Operators within one expression are generally executed from left to right, with the following additional rules:

3.3.5.1 According to the syntax given in section 3.3.1, the following rules of precedence hold:

first:	operators in string expressions, according to section 3.4
second:	sense
third:	↑
fourth:	× / ÷
fifth:	+ -

3.3.5.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently, the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6 Arithmetic of Real Quantities

Numbers and variables of type real must be interpreted in the sense of numerical analysis--i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. Nevertheless, no exact arithmetic is specified, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process being described, and is, therefore, expressed in terms of the language itself.

3.4 STRING EXPRESSIONS

3.4.1 Syntax

⟨primitive string expression⟩ ::= ⟨quotation⟩ | ⟨variable⟩ |
⟨function designator⟩ | (⟨string expression⟩)
⟨format expression⟩ ::= '⟨format⟩' | ⟨variable⟩ |
⟨function designator⟩ | (⟨string expression⟩)

⟨primary string expression⟩ ::= ⟨primitive string expression⟩ |
    quotation ⟨primitive string expression⟩
⟨secondary string expression⟩ ::= ⟨primary string expression⟩ |
    ⟨primary arithmetic expression⟩ text ⟨format expression⟩ |
    ⟨secondary string expression⟩ text ⟨format expression⟩ |
    ⟨secondary string expression⟩ sense ⟨format expression⟩
⟨simple string expression⟩ ::= ⟨secondary string expression⟩ |
    ⟨simple string expression⟩ ← ⟨secondary string expression⟩
⟨string expression⟩ ::= ⟨simple string expression⟩ | ⟨if clause⟩
    ⟨string expression⟩ else ⟨string expression⟩

3.4.2 Examples

```
'QUOTATION'
'-9.9(5)10-99'
'␣SSS␣'DOL'␣SS␣'CTS'␣'
DATE
MATERIAL [N - 1]
A[elements 1 : 4]
quotation M[element K]
TEXT (A, 'S(N)' )
if L then '1' else '0'
(if I < 5 then '9(4)+' else F)
T text F1 sense F2←T1
```

3.4.3 Semantics

A string expression is a rule for evaluating a string. A string value is an open quotation (see section 2.6, Quotations). Any string operator is undefined if the sequence of symbols generated by it is not an open quotation.

The principles of evaluating string expressions are completely analogous to the rules given in section 3.3.3 for arithmetic expressions.

Variables and function designators used as primitive string expressions or format expressions must be declared as type string. Such variables can have format expressions in their declaration (see sections 5.1 and 5.2).

3.4.4 Quotation Operator

The quotation operator involves converting an open quotation with an operand value into a quotation by means of including the open quotation within quotation marks.

3.4.5 Text Operator

The type of the left-hand operand must agree with the value of the right-hand operand, as shown below.

Left-hand operand type	Right-hand operand value
integer or real	numerical format
string	string format

The result of a text operation is a string representation of the value of its left-hand operand. If the right-hand operand has a numerical format, then the string value obtained is a blank quotation consisting of as many symbols as there are symbols in the format, excepting the symbols M, T, and U. If the right-hand operand has a string format, then the string value obtained is an open quotation. The

number of elements of that quotation is equal to the number of S, E, and ␣ symbols encountered in the value of the right-hand operand outside quotations that are inserts, plus the number of elements of such quotations (see section 2.7, Formats). The affected elements are considered to be numbered from left to right; this enumeration scheme defines the correspondence of the format elements (of individual symbols in particular) to the elements of the value of the obtained string.

3.4.6 Text Operator with a Numerical Format

The result of a text operator with a numerical format is obtained from the value of the first operand by means of removing from it the unaffected symbols (M, T, and U), and substituting symbols representing (roughly speaking) the numerical value of the left-hand operand in decimal notation for symbols other than space symbols; thus, space in a format indicates a required dispersal of the resulting string representation of a numerical value by the appropriate quantity of space symbols. For a numerical format with an exponent, the representation of the decimal number is maintained in normalized form with the exponent indicated.

Depending on the value of the right-hand operand, a decimal representation in a left-hand operand can be noted by inserting the symbols ␣ and * , adding the + and - signs, and removing the decimal point. This is more accurately described by the following rules.

3.4.6.1 The absolute value of a numerical quantity can be represented, as is well known, in the form of a sequence infinite at both ends and consisting of digits and a decimal

point. If two sequences have the same numerical value, then the sequence containing only a finite number of digits other than zeros is maintained. The left-most nonzero digit is called the highest-order digit. Zeros to the left of the highest-order digit are regarded as nonsignificant. A numerical value equaling zero is represented by a sequence in which all digits are zeros; they are all considered nonsignificant.

3.4.6.2 Congruence of Operand Positions. In the case of a numerical format without an exponent, the decimal representation of the value of the left-hand operand is equalized with the value of the right-hand operand according to the position of the decimal point, and then is rounded off or truncated (see sections 3.4.6.4 and 3.4.6.5). The sign of the number can be added in either at the beginning (see section 3.4.6.3) or at the end of the string representation of the numerical value.

In the case of a numerical format with an exponent, the decimal representation of the nonzero value of the left-hand operand is equalized with the value of the right-hand operand according to the highest-order digit--i.e., this digit is placed in the left-most digit position of the mantissa (see section 3.4.6.3); the representation of the mantissa is rounded off (which may require a new equalization) or truncated; the corresponding value of the exponent represented in the resulting string value positions matching the exponent format according to the integer representation rules is found. The symbol $_{10}$ is placed in the resulting string value position corresponding to itself in the format. If the value of the left-hand operand is equal to

zero, all digit positions of the mantissa and the exponent are filled with zeros.

It should be kept in mind that spaces should not be used in the value of a right-hand operand during equalization.

3.4.6.3 Suppressing Zeros, Suppressing and Positioning Signs. Each symbol P, * , or 9 in a format represents a separate digit position of the obtained string value; the symbols + and - in the sign part, except for the one at the far left, also represent separate digit positions of the obtained string value. The letter P denotes suppression of a nonsignificant zero and substitution by the space symbol; the symbol * denotes suppression of a nonsignificant zero and substitution by space symbols, except for the right-most zero for which the symbol + or - , or the ␣ (see below), is substituted. The digit 9 means that suppression can occur at no time in that position.

The sign of a number is placed in the resulting string value in accordance with the following rules:

If there is a + sign in the format (regardless of the format of the exponent), then in the appropriate position of the obtained string value is placed a + symbol when the value of the left-hand operand is non-negative, and a - symbol when the value is negative.

If there is a - sign in the format (regardless of the format of the exponent), then in the appropriate position of the obtained string value is placed the space symbol when the value of the left-hand operand is non-negative, and a - symbol when the value is negative.

If the sign is missing in the format (regardless of the format of the exponent), then the value of the left-hand

operand must be non-negative, or else the text operator is undefined.

3.4.6.4 Decimal Point. The position of the decimal point is shown in the format by the symbol . , the letter T, or the construction ⟨scale⟩.

In the first case, the decimal point is actually inserted in the corresponding character-position of the obtained string value.

In the second, the decimal point is implied in the string value--i.e., there is nothing designating it and its existence and location are defined only by the format.

The ⟨scale⟩ construction in a scale indicator points to the implied location of the decimal point as being to the right of the right-most digit position by as many positions as there are M symbols in the ⟨scale⟩ construction. In the case of a proper fraction scaled format, the construction ⟨scale⟩ points to the implied location of the decimal point as being to the left of the left-most digit position in the ⟨9 part⟩ construction by as many positions as there are M symbols in the ⟨scale⟩ construction. If in the latter case a sign part precedes the proper fraction scaled format, then all + or - signs, except the right-most one, are considered to be the same as the ␣ symbol.

If a numerical format contains neither the symbol . , the letter T, nor the ⟨scale⟩ construction, then the decimal point position is implied as being directly to the right of the right-most digit position, not counting the digit positions of the exponent.

3.4.6.5 Rounding off and Truncating. In transfers to a string value, a noninteger number is usually rounded

off; i.e., the nearest number to it that can be precisely represented in accordance with the given format is substituted (see, however, section 3.4.6.6). If though, there is a letter U in the format, "truncation" (elimination of the low-order character-positions) occurs, with the last-stored character being the right-most position, not counting the digit positions of the exponent.

3.4.6.6 Overflow. If the value of a left-hand operand according to its absolute value is too great to be represented in the form indicated by the right-hand operand, then the result of the text operator is undefined. If the value of an exponent is negative and too great according to its absolute value to be represented in the form indicated by the given exponent format, then the text operator is executed as if the value of the left-hand operand were zero.

3.4.7 Text Operator with a String Format

The execution of a text operator with a string format begins by ascertaining the correspondence, from left to right, of the E and S symbols, not belonging to inserts, of a right-hand operand value to elements of the left-hand operand value in an exponent. Where necessary, the value of the left-hand operand is supplemented on the right with a sufficient quantity of space symbols. The result of a text operator is obtained from the value of the right-hand operand by substituting in it for the symbols E and S, not belonging to inserts, the corresponding elements of the left-hand operand value and removing the outer quotation marks which are the insert elements. Each symbol S can be exchanged for an element consisting only of one quotation symbol.

3.4.8 Sense Operator in a String Expression

The execution of the sense operator with a string format begins by ascertaining the correspondence, from left to right, of all elements under consideration of a right-hand operand value (see section 3.4.5) to the elements of the value of the left-hand operand in an exponent. Where necessary, the value of the left-hand operand is supplemented on the right by a sufficient quantity of space symbols. The result of the sense operator is obtained from the value of the right-hand operand by substituting in it for the symbols E and S, not belonging to inserts, the corresponding elements of the value of the left-hand operand and removing the inserts. Each symbol S can be exchanged for an element consisting only of one quotation symbol.

3.4.9 Concatenation Operator

The result of the concatenation operator (←) is a sequence of symbols of the value of a left-hand operand followed by a sequence of symbols of the value of a right-hand operand.

3.4.10 Precedence of Operators

Operators within an expression are usually executed from left to right in accordance with the following additional rules.

3.4.10.1 According to the syntax given in section 3.4.1, the following precedence order is maintained:

first: quotation
second: text and sense
third: ←

3.4.10.2 The use of parentheses is interpreted in the sense given in section 3.3.5.2.

3.5 BOOLEAN EXPRESSIONS

3.5.1 Syntax

⟨relational operator⟩ ::= < | ≤ | = | ≥ | > | ≠

⟨arithmetic relation⟩ ::= ⟨simple arithmetic expression⟩
⟨relational operator⟩⟨simple arithmetic expression⟩

⟨string relation⟩ ::= ⟨simple string expression⟩
⟨relational operator⟩⟨simple string expression⟩

⟨relation⟩ ::= ⟨arithmetic relation⟩ | ⟨string relation⟩

⟨Boolean primary⟩ ::= ⟨logical value⟩ | ⟨variable⟩ |
⟨function designator⟩ | ⟨relation⟩ | (⟨Boolean expression⟩)

⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ | ¬ ⟨Boolean primary⟩

⟨Boolean monomial⟩ ::= ⟨Boolean secondary⟩ | ⟨Boolean monomial⟩
∧ ⟨Boolean secondary⟩

⟨Boolean term⟩ ::= ⟨Boolean monomial⟩ | ⟨Boolean term⟩
∨ ⟨Boolean monomial⟩

⟨implication⟩ ::= ⟨Boolean term⟩ | ⟨implication⟩
⊃ ⟨Boolean term⟩

⟨simple Boolean⟩ ::= ⟨implication⟩ | ⟨simple Boolean⟩
≡ ⟨implication⟩

⟨Boolean expression⟩ ::= ⟨simple Boolean⟩ | ⟨if clause⟩
⟨Boolean expression⟩ else ⟨Boolean expression⟩

3.5.2 Examples

```
PRICE = 15.5
A←B = 'SECTION␣5'
CHECK (K)
```

BALANCE + INCOME - EXPENDITURE > 0 ∧ BALANCE - NATURAL BALANCE = INVENTORY RESULT

FAMILY NAME = 'LEBEDEV'

SHOP ≠ 'FORGE' ∨ SHOP CODE ≠ '15'

DATE [M] > '18—05—64'

A ≤ '27.108' ∧ B [element I] > 'D'

if K < 1 then S > M else N ≤ R

if if if A then B else C then D else E then F else G ≠ 1

3.5.3 Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.5.4 Types

Variables and function designators entered as Boolean primaries must be declared Boolean (see sections 5.1, 5.2, and 5.5.4).

3.5.5 Operators

3.5.5.1 Arithmetic relations have the value true in the case where the corresponding relation is satisfied for the arithmetic expressions appearing in it; in the reverse case, they have the value false.

Checking the correctness of a string relation is based on lexicographic ordering of string values as produced by the following ordering of basic symbols (in ascending order):*

*This ordering is distinguished from the ordering of symbols of the GAIAPEI reference alphabet in that in the latter the letter Ъ follows the letter Я , and then comes the complete set of upper-case Latin-alphabet letters.

0 1 2 3 4 5 6 7 8 9 + - / , . ␣ 10 ↑ ( ) × = ; [ ] *
' ' ≠ < > : А Б В Г Д Е Ж З И Й К Л М Н О П Р С Т У
Ф Х Ц Ч Ш Щ Ы Ь Э Ю Я D F G I J L N Q R S U V W Z
‾ ≤ ≥ ∨ ∧ ⊃ ¬ ÷ ≡ % ◇ | — _ ! '' ъ ° ´ ← → ? ↓ ∅
± ∇

basic symbols, expressed as underscored words, in alphabetical order.*

In executing a relational operator on string values of different lengths, the shorter of them is regarded as being filled out on the right with enough space symbols to equalize the lengths.

3.5.5.2 The values of the logical operators ¬, ∧, ∨, ⊃, and ≡ are given by the following function table, in which T denotes true and F false:

B1	F	F	T	T
B2	F	T	F	T
¬ B1	T	T	F	F
B1 ∧ B2	F	F	F	T
B1 ∨ B2	F	T	T	T
B1 ⊃ B2	T	T	F	T
B1 ≡ B2	T	F	F	T

*The symbol represented here as '' appears to be equivalent to the ≫ symbol in section 2.2.3.--Trans.

3.5.6 Precedence of Operators

The operators within one expression are generally executed from left to right, with the following additional rules:

3.5.6.1 According to the syntax given in section 3.5.1, the following rules of precedence hold:

first: simple arithmetic expressions according to section 3.3.5, simple string expressions according to section 3.4.5.

second: $< \leq = \geq > \neq$

third: $\neg$

fourth: $\wedge$

fifth: $\vee$

sixth: $\supset$

seventh: $\equiv$

3.5.6.2 The use of parentheses is interpreted in the sense given in section 3.3.5.2.

3.6 DESIGNATIONAL EXPRESSIONS

3.6.1 Syntax

⟨label⟩ ::= ⟨identifier⟩

⟨switch identifier⟩ ::= ⟨identifier⟩

⟨switch designator⟩ ::= ⟨switch identifier⟩
[⟨subscript expression⟩]

⟨designational expression⟩ ::= ⟨label⟩ | ⟨switch designator⟩

3.6.2 Examples

```
P9
CHOOSE [N - 1]
SM [if Y < 0 then H else H + 1]
```

3.6.3 Semantics

A designational expression is a rule for obtaining a label of a statement (see section 4, Statements). Again, the principles of evaluation are entirely analogous to the rules given for arithmetic expressions (section 3.3.3). A switch designator refers to the corresponding switch declaration (see section 5.4, Switch Declarations), and by the actual numerical value of its subscript expression selects one of the labels comprising the switch list in the switch declaration. This value serves as the number of a label on the switch list, reading from left to right.

3.6.4 The Subscript Expression

The evaluation of the subscript expression is analogous to that of subscripted variables (see section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3,...,n, where n is the number of entries in the switch list.

3.7 SUBARRAY DESIGNATORS

3.7.1 Syntax

⟨subscript scale element⟩ ::= ⟨subscript expression⟩ |
⟨subscript expression⟩ : ⟨subscript expression⟩
⟨subscript scale catalog⟩ ::= ⟨subscript scale element⟩ |
⟨subscript scale catalog⟩ , ⟨subscript scale element⟩
⟨subscript scale⟩ ::= ⟨subscript scale element⟩ |
((subscript scale catalog))

⟨subscript scale list⟩ ::= ⟨subscript scale⟩ |
⟨subscript scale list⟩ , ⟨subscript scale⟩
⟨subarray designator⟩ ::= ⟨array identifier⟩
[⟨subscript scale list⟩] | ⟨array identifier⟩
[⟨subscript scale list⟩ element ⟨position list⟩]

3.7.2 Examples

```
A[K, 10 : N]
P[(1, 3, 6 : 15), (S : T + 1, U)]
TM[1 : 10 elements 2, 4, 6, 8]
```

3.7.3 Semantics

The subarray designator defines some ordered subset of array components (subscripted variables). This subset consists of all components, each subscript of which has a value permissible according to the subscript scale. The subarray designator can be used outside a compound designator (see section 3.8) only if the array identifier is a first-level identifier (see section 5).

3.7.4 Subscript Scale

The number of subscript scales in a scale list must equal the array dimension. Each subscript scale element either is a subscript expression and defines a permissible value equal to an integral value of that expression (see section 3.1.4.2), or consists of two subscript expressions separated by a colon; in the latter case, sequential integral values, beginning with the value of the first subscript expression and ending with the value of the second subscript

expression in the pair, are permissible. If the value of the second subscript expression is less than the value of the first, then that subscript scale element defines an empty set of permissible values. If the subscript scale catalog consists of more than one element, then it is obligatory that it be enclosed in parentheses. The subscript scale is defined only if it defines at least one permissible value, if all permissible values are within the appropriate bound pair according to the array declaration, and if all permissible values defined by the scale catalog in the order of the sequence of the elements from left to right reflect a strictly increasing sequence. The subarray designator is undefined if just one of the subscript scales is undefined.

3.7.5 Subarray Length

The total number of permissible subscript values, determined by the appropriate subscript scale, is termed the length of the subarray's individual measurement. As in the case of an array, the product of the lengths of all measurements is called the subarray length.

3.7.6 Ordering of Subarray Components

The order of subarray components is determined by the ordering of the components of the array from which the given subarray is extracted (see section 5.2.7).

3.7.7 Position List

A position list can occur in a subarray designator only if the array is declared as string or the format of the array

components is indicated in the array declaration. Thus, as in the case of a variable, the position list indicates which elements of the string value of an array component without a position list produce the values of the appropriate component of the given subarray.

3.7.8 Complete Subarray Designator

A subarray designator is considered complete if every value that the subscript can take on from among the current values of the subscript bounds (see sections 3.1.4.2 and 5.2.3.1) is allowable according to the appropriate subscript scale.

3.8 CONSTITUENT DESIGNATORS

3.8.1 Syntax

⟨compound name⟩ ::= ⟨simple compound identifier⟩ |
⟨compound-array identifier⟩ | ⟨compound-array identifier⟩
[⟨subscript scale list⟩]
⟨composition element⟩ ::= ⟨variable-constituent tail⟩ |
⟨array identifier⟩ | ⟨subarray designator⟩ |
⟨compound designator⟩
⟨composition list⟩ ::= ⟨composition element⟩ |
⟨composition list⟩ , ⟨composition element⟩
⟨composition⟩ ::= ⟨composition element⟩ | (⟨composition list⟩)
⟨compound designator⟩ ::= ⟨compound name⟩ |
⟨compound name⟩ . ⟨composition⟩
⟨constituent designator⟩ ::= ⟨compound designator⟩

3.8.2 Examples

A

X[I]

BI.(B, C, D)

M.(B[elements 1 : 15], C[H,(1, 4 : 6)], D. T)

H[1 : 4]. (F, G, R[L : NR], H[K elements 3 : 8, 11])

3.8.3 Semantics

The constituent designator defines some ordered subset of primary components (variable-constituents) of a compound (see section 5.3, Compound Declarations). The constituent designator begins with that compound's identifier, which must also be the identifier of the first level in the corresponding declaration. Compound designators, in contrast to constituent designators, can begin with higher-level identifiers; in such a case, the level of the initial identifier is considered also as the level of the compound designator. A compound designator having a level higher than the first cannot have a defined sense if it is examined separately from a compound designator of a level lower than that at which it is located as a composition element. This accounts for why, at levels higher than the first, different elements of the same or of several compounds can be designated by identical identifiers. The remaining syntax and semantics for the compound designator and the constituent designator are the same.

3.8.4 Compound Designator Value

3.8.4.1 If the compound designator consists only of a simple compound identifier, then the set of values of all elements of the compound structure specified by that identifier serves as its value (see section 5.3).

3.8.4.2 If the compound designator has the form of a compound-subarray identifier--that is, if it consists of a compound-array identifier accompanied by a subscript scale list enclosed in square brackets--then the subset of values of the compound-array components separated out according to the same rules as for defining the value of a subarray designator (see section 3.7, Subarray Designators) serves as its value. The set of values of all elements of the structure of the given compound array, enumerated in its declaration, constitutes the values of each component of the compound-array. A compound name consisting only of a compound-array identifier denotes the same thing as a compound name written in the form of a complete compound-array designator (see section 3.7.8) with the same identifier.

3.8.4.3 A composition included within a compound designator explicitly enumerates the structure elements (variable-constituents, subarrays, arrays, and compounds) of the appropriate compound whose values form the values of the given compound designator (if it is a simple compound) or the values of each component of a compound array.

3.8.5 Compound Designator Component Ordering

Composition elements in a composition list must be arranged so that their initial identifiers occur in the same order in which they are found in the declaration of the structure of the corresponding compound (see section,

5.3, Compound Declarations). Thus, the ordering of the primary components of a compound designator must fully coincide with their ordering within the compound as a whole.

3.8.6 Compound Designator Abbreviation

A compound designator of the form

N. I1. I2.....IK. CD

or

N. I1. I2.....IK. CE

(where N is a compound name; I1, I2,...,IK are simple compound or compound-array identifiers; CD is a compound designator; and CE is a composition element) can be written in short form as

N. CD

or, correspondingly,

N. CE .

The abbreviated notation is equivalent to the complete notation if there is no possibility of ambiguity arising as a result of the abbreviation, and if and only if a compound designator of the form

N. CD1

or

N. CE1

(with the same compound name N and having an initial compound designator identifier CD1 or composition element identifier CE1 matching the initial identifier CD or CE)

can arise only when the full notations of the compound designators

N. I1. I2.....IK. CD1

or

N. I1. I2.....ID. CE1

are abbreviated.

Using the same notation, the composition element

I1. I2.....IK. CD

or

I1. I2.....IK. CE

in the compound designator

N. (..., I1. I2.....IK. CD,...)

or, respectively,

N. (..., I1. I2.....IK. CE,...)

can be abbreviated in analogous situations as

CD

or, correspondingly,

CE .

3.8.7 Examples

Given are compounds with the following declarations: compound A. (string B; integer C; compound D. (integers E, C, G)); compound array M[1 : 5]. (string B; compound C. integer E; compound D. (integers E, F); integer array G[1 : 2]).

Let the current values of these compounds be defined by the following tables:

A				
B	C	D		
		E	C	G
M	2	10	20	30

	M					
	B	C	D		G	
		E	E	F	1	2
1	T	4	21	14	40	16
2	E	3	22	13	30	17
3	K	2	23	12	20	18
4	S	5	24	15	50	19
5	T	1	25	11	10	29

Then, some possible constituent designators define the following sets of values:

Constituent Designator	Ordered Set of Values	Remarks
A.	M, 2, 10, 20, 30	
A.B	M	
A.C	2	
A.(C, B)	not defined	see section 3.8.5
A.D	10, 20, 30	
A.(C, D. C)	2, 20	
A.(B, C, D.(E, G))	M, 2, 10, 30	
M.[1]	T, 4, 21, 14, 40, 16	
M.C	4, 3, 2, 5, 1	
M.C.E	4, 3, 2, 5, 1	
M.D	21, 14, 22, 13, 23 12, 24, 15, 25, 11	
M.E	not defined	see section 3.8.6
M[3].B	K	
M[5].G	10, 29	
M[2 : 3].(B, F, G[2])	E, 13, 17, K, 12, 18	
M[(1, 4)].(D. E, G)	21, 40, 16, 24, 50, 19	

3.9 MULTICOMPONENT EXPRESSIONS

3.9.1 Syntax

⟨primary multicomponent expression⟩ ::= ⟨array identifier⟩ |
⟨subarray designator⟩ | ⟨constituent designator⟩ |
(⟨multicomponent expression⟩)

⟨nondesignational expression⟩ ::= ⟨arithmetic expression⟩ |
⟨Boolean expression⟩ | ⟨string expression⟩

⟨secondary multicomponent expression⟩ ::=
⟨primary multicomponent expression⟩ | ⟨operator⟩ |
⟨primary multicomponent expression⟩ |
⟨nondesignational expression⟩

⟨simple multicomponent expression⟩ ::=
⟨secondary multicomponent expression⟩ |
⟨simple multicomponent expression⟩ | ⟨operator⟩ |
⟨secondary multicomponent expression⟩

⟨multicomponent if clause⟩ ::= if ⟨multicomponent expression⟩
then

⟨multicomponent expression⟩ ::=
⟨simple multicomponent expression⟩ | ⟨if clause⟩
⟨multicomponent expression⟩ else ⟨multicomponent expression⟩ |
⟨multicomponent if clause⟩⟨multicomponent expression⟩
else ⟨multicomponent expression⟩

3.9.2 Examples

A × X[I : N] + B × Y[I : N]

M[I : P].(C, D) - K[I : P,(2, 6)]

C[I : 20] < 0

P ≥ - Q[-K : K] ∧ ¬ KSI[0 : 2 × K]

```
if X > Y then AX[I : 200 elements 1 : 15] else AY
T[1 : 5] ← TI[2 : 6] ← '.'
if A[I : T]. L then A[I : T], B else A[I : T]. C
```

3.9.3 Semantics

The multicomponent expression is a rule for computing an ordered set of numbers, logical values, or strings. If the multicomponent expression is a constituent designator, then its components can be of different types; in all other cases, they must be of the same type.

An array identifier in the role of a primary multicomponent expression means the same as a complete subarray indicator with that identifier (see section 3.7.8).

An operator can be used in a secondary multicomponent expression only when the given operator can be applied to the value of every primary component of the operand--i.e., of the primary multicomponent expression in front of which the operator stands. (It follows that this operator can be only - or ¬ , or quotation.) Such a secondary multicomponent expression is evaluated as a result of the actual execution of this operator over each primary component of the operand.

In a simple multicomponent expression, the only operands that can be associated with the operator are those between any primary component of the first operand and the corresponding component of the second operand for which the given

operator can be executed. The constituent designator can be an operand of a multicomponent expression (either simple or secondary) if all of its primary components have precisely the same type.

A simple multicomponent expression containing an operator is undefined if the operands do not contain the same quantity of primary components. However, if one of the operands is a nondesignational expression, then it is regarded as a multicomponent expression in which the value of the nondesignational expression is repeated as many times as there are primary components containing the other operand. Operators specified in a simple multicomponent expression are executed over each pair of appropriate primary components of both operands. The resulting value forms the values of the simple multicomponent expression.

The rules for evaluating a multicomponent expression that includes within itself an ordinary if clause are completely analogous to the rules for evaluating arithmetic expressions (see section 3.3.3). All primary components of a multicomponent expression in a multicomponent if clause must be of type Boolean, and their number must be the same as the number of primary components of the multicomponent expressions following the else symbol and up to the multicomponent if clause. A true value of a primary component of a multicomponent expression in a multicomponent if clause causes the value of the primary component of the expression following the multicomponent if clause to be selected as the value of the corresponding component of the entire multicomponent expression containing the multicomponent

if clause; in the case of a false value, the primary component of the expression standing after the else delimiter is selected.

Precedence of operators and the use of parentheses in multicomponent expressions conforms to the same rules as for arithmetic, string, and Boolean expressions (see sections 3.3.5, 3.4.5, and 3.5.6).

4. STATEMENTS

The units of operation within the language are called statements. They are normally executed consecutively as written. However, this sequence of operations can be broken by go to statements which define their successor explicitly, and shortened by conditional statements which can cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements can be provided with labels.

Since sequences of statements can be grouped together into compound statements and blocks, the definition of statement must necessarily be recursive. Also, since declarations, considered in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1 COMPOUND STATEMENTS AND BLOCKS

4.1.1 Syntax

⟨unlabeled basic statement⟩ ::= ⟨assignment statement⟩ |
⟨go to statement⟩ | ⟨dummy statement⟩ |
⟨procedure statement⟩

⟨basic statement⟩ ::= ⟨unlabeled basic statement⟩ |
⟨label⟩ : ⟨basic statement⟩

⟨unconditional statement⟩ ::= ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩

⟨statement⟩ ::= ⟨unconditional statement⟩ |
⟨conditional statement⟩ | ⟨for statement⟩

⟨compound statement tail⟩ ::= ⟨statement⟩ end | ⟨statement⟩ ;
⟨compound statement tail⟩

⟨block head⟩ ::= begin ⟨declaration⟩ | ⟨block head⟩ ;
⟨declaration⟩

⟨unlabeled compound statement⟩ ::= begin
⟨compound statement tail⟩

⟨unlabeled block⟩ ::= ⟨block head⟩ ;
⟨compound statement tail⟩

⟨compound statement⟩ ::= ⟨unlabeled compound statement⟩ |
⟨label⟩ : ⟨compound statement⟩

⟨block⟩ ::= ⟨unlabeled block⟩ | ⟨label⟩ : ⟨block⟩

⟨program⟩ ::= ⟨block⟩ | ⟨compound statement⟩

If arbitrary statements, declarations, and labels are denoted by the letters S, D, and L, respectively, the basic syntactic units can be illustrated as follows:

Compound statement:

L : L : ... L : begin S; S; ...; S end

Block:

L : L : ... L : begin D; D; ...; D; S; S; ...; S end

It should be kept in mind that each of the statements S can again be a compound statement or block.

4.1.2 Examples

Basic statements:

```
A := P + K
to LABEL M
START: CONTINUE: L := 7.993
```

Compound statement:

```
begin X := 0; for Y := 1 : N
  do X := X + A[Y]; if X > K then
  to STOP else if X > B - 2 then to C;
  AB : ST : B := X + B4
end
```

Block:

```
K : begin integer I, K; real B;
      for I := 1 : M do
      for K := I + 1 : M do
      begin B := A[I,K]; A[I,K] := A[K,I];
        A[K,I] := B
      end FOR I, K.
    end BLOCK K.
```

4.1.3 Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block can, through a suitable declaration (see section 5, Declarations), be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the given block has no existence outside the block; and (b) that any entity represented by this identifier outside the given block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block are non-local to it--i.e., they represent the same entity both inside the block and in the level immediately outside

it. A label separated by a colon from a statement--i.e., labeling that statement--behaves as though declared in the head of the smallest embracing block; i.e., the smallest block whose brackets begin and end enclose that statement. In this context, a procedure body must be considered as if it were enclosed by begin and end and treated as a block.

Since a statement in a block can also itself become a block, the concepts local and non-local to a block must be understood recursively. Thus, an identifier that is non-local to a block A can be local or non-local to the block B for which A is one of its statements.

4.2 ASSIGNMENT STATEMENTS

4.2.1 Syntax

⟨left part⟩ ::= ⟨variable⟩ := | ⟨procedure identifier⟩ :=
⟨left part list⟩ ::= ⟨left part⟩ | ⟨left part list⟩⟨left part⟩
⟨multicomponent left part⟩ ::= ⟨array identifier⟩ := |
⟨subarray designator⟩ := | ⟨constituent designator⟩ :=
⟨assignment statement⟩ ::= ⟨left part list⟩
⟨nondesignational expression⟩ |
⟨multicomponent left part⟩⟨multicomponent expression⟩ |
⟨multicomponent left part⟩⟨nondesignational expression⟩

4.2.2 Examples

```
S := P[0] := N := N + 1 + S
S[Y, K + 2] := 3 - ARCTG(P × ZETA)
V := Q > Y ∧ Z
DATE := '19␣MAY␣1964'
```

```
SA[1] := 'FOUNDRY'
A. B.[I] := C := D + E - F(R)/K. X[M]
N[I, 1 : I - 1] := N[1 : I - 1, I]
D[1 : 4]. (K, L[elements 1 : 5]) := G[1 : 2, 1 : 2].
  (I, L[elements 2 : 6])
S[I, 1 : N] := S[I, 1 : N] + B[J, 1 : N] × A[I, J]
I[1 : S, 1 : T]. G := 1
A[1 : 20]. (B. D[6], C[1 : 2]) := E[1 : 5, (3, 6, 9), 0 : 3]
SUMMARY. (WEIGHT, SUM) := SUMMARY. (WEIGHT, SUM) + (if
   ACCOUNT [I]. SHOP = SA[1] ∧ ACCOUNT [I].
   SECTION = SA[2] then ACCOUNT I. (WEIGHT, SUM) else 0)
```

4.2.3 Semantics

Assignment statements serve for assigning the value of some expression to one or several variables or procedure identifiers or to a multicomponent value. Assignment to a procedure identifier can only occur within the body of a procedure defining the value of a function designator (see section 5.5.4).

4.2.4 Assignment to Single-Component Quantities

If the quantities in left parts are single components, execution of the assignment statement must in the general case take place in three steps as follows:

4.2.4.1 All subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.4.2 The nondesignational expression producing the right part of the statement is evaluated. Where necessary, the corresponding transfer function is applied (see section 4.2.7).

4.2.4.3 The value of the nondesignational expression is assigned to all left part variables with subscript values evaluated in step 4.2.4.1.

4.2.5 Assignment to a Multicomponent Quantity

If the quantity in the left part of a statement is multicomponent, the assignment statement is executed in the general case in three steps:

4.2.5.1 All subscript expressions of the left part are evaluated sequentially, thereby separating out an ordered set of primary array components or a list.

4.2.5.2 The multicomponent or nondesignational expression in the right part of the statement is evaluated.

4.2.5.3 If the expression in the right part is multicomponent, the value of each primary component is assigned to the appropriate component of the left part. The correspondence of primary components is established solely in terms of their ordering (see sections 5.2.7 and 5.3.6). Where necessary, in each separate assignment process the corresponding transfer function is applied (see section 4.2.7).

If the right part of the statement defines a single value, it is possible for that value to be assigned to all primary components of the left part after applying the required transfer function.

4.2.6 Size and Composition Agreement

If both parts of an assignment statement are multicomponent quantities, agreement of array sizes and compound compositions must be observed.

4.2.6.1 Under agreement of array sizes (including compounds) is understood equality of lengths (see section 5.2.3.2). Individual measurement lengths, as with the dimensions of an array, may not coincide.

4.2.6.2 Agreement of compound compositions is established recursively beginning with the first level. Compositions are considered identical if the number of compound elements coincide and if the composition elements located at the same places in composition lists correspond to each other according to size and make-up.

4.2.6.3 A constituent designator in one part of an assignment statement can be combined with a subarray designator in another part only in the case where the total number of primary designator components matches the subarray length.

4.2.6.4 An array identifier in an assignment statement left part is understood as a complete subarray designator with that identifier (see section 3.7.8).

4.2.7 Types and Formats

The same type must apply to all variables and procedure identifiers in a left part list. The type assigned to a procedure identifier is given by the declaration appearing as the first symbol of the corresponding procedure declaration (see section 5.5.4). The type of a variable (or of a primary component of a multicomponent quantity) in a left part must agree with the type of the expression or component value assigned to it from the right part. This means:

4.2.7.1 If the left part variable is of type Boolean, the value determined by the right part must be of type Boolean or string. In the latter case, the symbol 0, setting the assigned value false, or the symbol 1, setting the value true, must serve as this value; otherwise, the assignment statement is undefined.

4.2.7.2 If the left part variable type is real or integer, the value defined by the right part can be one of the types real, integer, or string. If the types of the left and right parts are not the same, the appropriate transfer function is automatically applied. For a transfer from type real to type integer, the transfer function is considered to produce a result equivalent to

ENTIER(A + 0.5) ,

where A is the value of an arithmetic expression or of a multicomponent expression primary component.

Transfer from type string to one of the numerical types, if the left part variable has a format expression F in its declaration, is accomplished using a transfer function giving a result equivalent to

T sense F ,

where T is the value of a string expression or of a multicomponent expression primary component. If, though, the format of the left part variable is not declared, the value of the string expression is free of the editing symbols (␣ and *). The obtained text must be a number in the sense of section 2.5, and, in addition, a transfer from type integer to real or vice versa can be required only according to the rules of the preceding paragraph.

4.2.7.3 If a left part variable has type string, the corresponding component of the right part must also define a value of type string; i.e., it must be either, on the one hand, a string or a Boolean expression or, on the other, a variable in whose declaration a format expression is given (see section 5.1.3.2).

If, in addition, a format expression F is contained in the declaration of a left part variable, the transfer function must produce a result equivalent to

S text F ,

where S is a string value of a nondesignational expression or of a multicomponent expression primary component.

4.2.8 The Role of the Position List

If a left part variable is equipped with a position list, assignment of new values is limited to those elements defined by that list. The values of the remaining elements of such a variable do not change as the result of the execution of an assignment statement.

4.3 GO TO STATEMENTS

4.3.1 Syntax

⟨go to statement⟩ ::= to ⟨designational expression⟩

4.3.2 Examples

```
to M8
to EXIT [N + I]
to SM [if Y < 0 then N else N + I]
```

4.3.3 Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus, the next statement executed will be the one having this value as its label.

4.3.4 Restriction

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement can, however, lead from outside into a compound statement.

4.3.5 Go To with an Undefined Switch Designator

A go to statement is undefined if the designational expression is a switch designator whose value is undefined.

4.4 DUMMY STATEMENTS

4.4.1 Syntax

⟨dummy statement⟩ ::= ⟨empty⟩

4.4.2 Examples

```
B:
begin ...; FINISH: end
```

4.4.3 Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5 CONDITIONAL STATEMENTS

4.5.1 Syntax

⟨if clause⟩ ::= if ⟨Boolean expression⟩ then

⟨unconditional statement⟩ ::= ⟨basic statement⟩ |
⟨compound statement⟩ | ⟨block⟩

⟨if statement⟩ ::= ⟨if clause⟩⟨unconditional statement⟩

⟨conditional statement⟩ ::= ⟨if statement⟩ |
⟨if statement⟩ else ⟨statement⟩ | ⟨if clause⟩
⟨for statement⟩ | ⟨label⟩ : ⟨conditional statement⟩

4.5.2 Examples

```
if X > 0 then N := N + 1

if B > Y then M := N + D else to R

if S < 0 ∨ R then AA :
begin if X < B then A := B/C
  else Y := 2 × A
end
else if B > C then A := B - X
else if B < C - 1 then to CI
```

4.5.3 Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1 If Statement. An unconditional statement appearing in an if statement is executed if the Boolean expression appearing in the if clause is true. Otherwise, it is skipped and the operation continues with the next statement.

4.5.3.2 Conditional Statement. According to the syntax, two different forms of conditional statements are possible. These can be illustrated as follows:

if B1 then S1 else if B2 then S2 else S3 ; S4

if B1 then S1 else if B2 then S2 else if B3 then S3 ; S4 .

Here B1, B2, B3 are Boolean expressions, while S1, S2, S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement can be described as follows: The Boolean expressions of the if clauses are evaluated in sequence from left to right. The evaluation continues until one yielding the value true is found. Then the unconditional statement immediately following this Boolean expression is executed. If this statement does not define its successor explicitly, the next statement executed will be the statement following the complete conditional statement (S4 in the examples above). Thus, the effect of the delimiter else can be described by saying that it defines the successor of the statement it (the delimiter) follows to be the statement following the complete conditional statement.

In accordance with the sense of the preceding paragraph, the construction

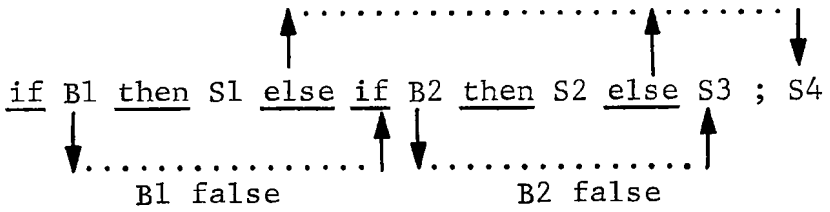
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩ .

If none of the Boolean expressions appearing in the if clauses is true, the effect of the entire conditional statement is equivalent to that of a dummy statement.

For further explanation, the following picture may be useful:



4.5.4 Go To into a Conditional Statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of the else delimiter.

4.6 FOR STATEMENTS

4.6.1 Syntax

⟨for list element⟩ ::= ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ step ⟨arithmetic expression⟩
until ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ : ⟨arithmetic expression⟩ |
⟨arithmetic expression⟩ while ⟨Boolean expression⟩

⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩ ,
⟨for list element⟩

⟨for clause⟩ ::= for ⟨variable identifier⟩ := ⟨for list⟩ do

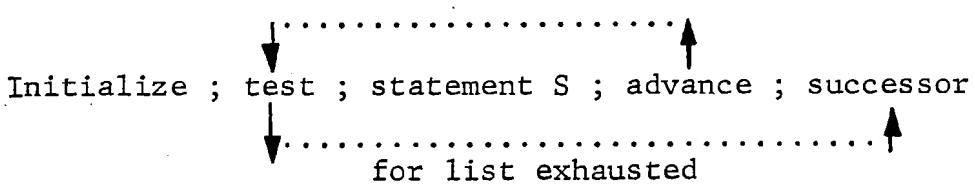
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩ |
⟨label⟩ : ⟨for statement⟩

4.6.2 Examples

```
for Q := 1 step P until N do A[Q] := B[Q]
for K := 1, V1 × 2 while V1 < N do
for J := 1 + G, L, 1 : M, C + D do E[K, J] := F[K, J]
```

4.6.3 Semantics

A for clause causes the statement S following it to be repeatedly executed several times or not at all. In addition, it performs a sequence of assignments of a value to a simple variable, the identifier of which is indicated in the given clause. Ultimately, this variable will be designated the "for parameter." The process can be explicated by the following picture:



In this picture, the word "initialize" means: perform the first assignment of the for clause. "Advance" means: perform the next assignment of the for clause. "Test" means: if the for clause permits another assignment, go to the execution of the S following the for clause; if it doesn't, continue program execution with the statement following the for statement.

4.6.4 The For List Elements

The for list gives a rule for obtaining the values which are consecutively assigned to the for parameter. This sequence of values is obtained from the for list elements by taking them sequentially in the order in which they are written. The sequence of values generated by each of the four kinds of for list elements and the corresponding execution of the statement are given by the following rules.

4.6.4.1 Arithmetic Expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2 Arithmetic Progression Element. An element of the form A step B until C, where A, B, and C are arithmetic expressions, gives rise to an execution which can be described most concisely in terms of additional statements as follows:

```
      V := A ;
L1 : if (V - C) × SIGN(B) > 0 then to ELEMENT EXHAUSTED;
      S ;
      V := V + B ;
      to L1 ;
```

where V is the for parameter and "ELEMENT EXHAUSTED" points to the evaluation according to the next element in the for list, or if the given arithmetic progression element is the last on the list, to the next statement in the program.

4.6.4.3 Arithmetic Progression Element at the One Step. An element of the form A : C, where A and C are arithmetic expressions, is treated as an element of the form A step 1 until C. Here, the for parameter must have an integer type declaration.

4.6.4.4 Iteration Element. The execution governed by a for list element of the form E while F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional statements as follows:

```
L3 : V := E ;
     if ¬ F then to ELEMENT EXHAUSTED;
     S ;
     to L3 ;
```

where the notation is the same as in 4.6.4.2.

4.6.5 The Value of the For Parameter After Exit

After exit from the statement S (supposed to be compound) via some go to statement, the value of the for parameter will be the same as it was immediately prior to execution of the go to statement.

If, on the other hand, the exit is due to exhaustion of the for list, the value of the for parameter is undefined after the exit.

4.6.6 Go To Statement Leading into a For Statement

The effect of a go to statement outside a for statement and referring to a label inside the for statement is undefined.

4.7 PROCEDURE STATEMENTS

4.7.1 Syntax

⟨procedure statement⟩ ::= ⟨procedure identifier⟩
⟨actual parameter part⟩

4.7.2 Examples

```
SPUR (A) ORDER: (7) RESULT TO: (V)
TRANSPOSE (W, V + 1)
```

```
ABSMAX (A, N, M, Y, I, K)
INNERPRODUCT (A[T, P, U], B[P], 10, P, Y)
SELECT (M[I]. (B, F), 5, K[NK]. (E, F[1 : 2], NK,
  I, M[I]. B = 'T')
```

These examples correspond to the examples given in section 5.5.2. If a compound array M is declared as in section 3.8.7 and with the component values indicated there, and a list K has the declaration compound array K[1 : integer NK]. (integer D; integer array F[1 : 2]), then the effect of execution of the above-presented SELECT procedure statement is for the variable NK to assume the value 2 and the array components of K the values shown in the following table:

K			
	D	F	
		1	2
1	4	40	16
2	1	10	29

4.7.3 Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (see section 5.5, Procedure Declarations). The effect of this execution should be equivalent to the effect of the following operations on the program at the time of execution of the procedure statement.

4.7.3.1 Value Assignment (Call by Value). All formal parameters included in the value list on the procedure declaration heading are assigned the values of the corresponding actual parameters; these assignments are considered

as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (see section 5.5.5). As a consequence, variables called by value should be considered as non-local to the body of the procedure, but local to the fictitious block (see section 5.5.3).

4.7.3.2 Name Replacement (Call by Name). Any formal parameter not included in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body are avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3 Body Replacement and Execution. Finally, the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator are avoided through suitable systematic changes of the latter identifiers.

4.7.4 Actual-Formal Parameter Correspondence

Correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. Correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5 Restrictions

For a procedure statement to be defined, it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGEC statement.

This poses the restriction on any procedure statement that the kind, structure, type, and format of each actual parameter be compatible with the kind, structure, type, and format of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1 Only a variable, a subarray designator, or a constituent designator (the particular occurrences of an expression), in addition to an array identifier that is understood to be a complete subarray designator (see section 4.2.6.4), can correspond as an actual parameter to a formal parameter occurring in the form of a left part value of an assignment statement within a procedure body and not called by value.

4.7.5.2 If a formal parameter called by name is used in a procedure body as an array identifier of specified dimension, then to it must correspond as the actual parameter either an array identifier of the same dimension, or a constituent designator of the form

I1. I2.....IK. IA ,

where the I1, I2,...,IK are explicit or implied (see section 3.8.6) simple compound identifiers, and IA is an array identifier of the same dimension as the array established by the formal parameter.

4.7.5.3 If a formal parameter used in a procedure body as an array identifier of specified dimension is called by value, and the corresponding actual parameter satisfies the requirements of the preceding section, then the local array produced in the course of the execution takes on the same subscript bounds as the actual array. But if the array established in the procedure body by the formal parameter is one-dimensional, then an arbitrary multicomponent expression (specifically, a constituent designator), all primary components of which have the same type, is also allowable as the corresponding actual parameter. Then, the local array produced in the course of the call by value takes on, if the if clause of section 4.7.5.2 has not been executed, a lower subscript bound equaling 1, and an upper bound equaling the number of primary components of the multicomponent expression.

4.7.5.4 A procedure identifier or a switch identifier, because these latter do not possess values, cannot in general correspond to a formal parameter called by value. (The exception is the procedure identifier of a procedure

declaration which has an empty formal parameter part (see section 5.5.1) and which defines the value of a function designator (see section 5.5.4). Such a procedure identifier is in itself a complete expression.)

4.7.5.5 Any formal parameter can place restrictions on the type and format of both the corresponding actual parameter associated with it and its primary components (these restrictions can be completely or partially given through specifications in the procedure heading, and also derive from the nature of the use of a formal parameter in a procedure body). In the procedure statement such restrictions must evidently be observed.

4.7.5.6 A procedure can refer to itself neither during execution of the procedure body statements, nor during evaluation of the actual parameters to which the formal parameters called by name correspond, nor during evaluation of the statements occurring in the declarations within the procedure body.

4.7.6 Parameter Delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected, beyond their number being the same. Thus, the information conveyed by using parameter delimiters is entirely optional.

4.7.7 Standard Procedure Statements

Among the fixed identifiers, it is recommended that there be a LIBRARY identifier, serving to fetch library

subroutines. The procedure operator for this purpose has the following form:

LIBRARY (⟨quotation⟩ , ⟨actual parameter list⟩) ,

where the value of the actual parameter ⟨quotation⟩ is the name of the library procedure, and the actual parameter list consists of actual parameters for the processing of this procedure. All delimiters placed on this list are defined only by the nature of the library concerned and can be outside the scope of the ALGEC reference language.

5. DECLARATIONS

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid only for one block. Outside this block, the given identifier can be used for other purposes (see section 4.1.3).

Dynamically, this implies the following: at the time of an entry into a block (through the begin, since the labels inside are local and, therefore, inaccessible from outside), all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers have already been defined by other declarations outside the block, they are for the time being given a new significance. On the other hand, identifiers not declared for the block retain their old meaning.

At the time of an exit from a block (through end or by a go to statement) all identifiers declared for the block lose their local significance.

Apart from labels, formal parameters of procedures, and, possibly, identifiers of standard functions (see section 3.2.4) and standard procedures (see section 4.7.7), all identifiers of a program must be declared. No identifier should be declared more than once in a block, except in the case where it is the identifier of a list structure element (see section 5.3.5).

Identifiers introduced by any declaration that is not an element of a declaration of a structure (see section 5.3) are considered first-level identifiers.

Syntax

⟨declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ |
⟨compound declaration⟩ | ⟨switch declaration⟩ |
⟨procedure declaration⟩

5.1 TYPE DECLARATIONS

5.1.1 Syntax

⟨type list element⟩ ::= ⟨variable identifier⟩ |
⟨variable identifier⟩ format ⟨format expression⟩
⟨type list⟩ ::= ⟨type list element⟩ | ⟨type list element⟩ ,
⟨type list⟩
⟨type⟩ ::= real | integer | string | Boolean
⟨variable identifier designator⟩ ::= ⟨variable identifier⟩ |
⟨compound identifier⟩ . ⟨variable identifier designator⟩
⟨variable prototype designator⟩ ::=
⟨variable identifier designator⟩
⟨variable identifier list⟩ ::= ⟨variable identifier⟩ |
⟨variable identifier list⟩ , ⟨variable identifier⟩
⟨type declaration⟩ ::= ⟨type⟩⟨type list⟩ | as
⟨variable prototype designator⟩ : ⟨variable identifier list⟩

5.1.2 Examples

```
integers A, B, C format '9(4)+'
Booleans OVERFLOW, B, END
strings ORDER, DATE, ADDRESS format 'C(A)', PART format
  'C(B)', CODE format 'CCE', SHOP format 'CC'
as SHOP: FORM OF PAYMENT, OPERATION, WAREHOUSE
```

5.1.3 Semantics

5.1.3.1 Type declarations serve to declare certain identifiers to represent simple variables of a given type. Variables declared as real can only assume positive or negative values, including zero. Variables declared as integer can only assume positive and negative integral values, including zero. Variables declared as string can assume only the values of open quotations. Variables declared as Boolean can assume only the values true and false.

In arithmetic expressions, any position that can be occupied by a variable declared as real can be occupied by a variable declared as integer.

5.1.3.2 When a format expression is explicitly declared, it refers only to that quantity with whose identifier it is associated by the format declaration. It is treated in accordance with the rules of sections 2.7 and 3.4.

Assignment of a numerical format for integer and real type variables means that such a variable in addition to its principal value, has also a string value that is a blank quotation of the indicated format. A string value consisting of one of the symbols 0 (corresponding to the value false) or 1 (corresponding to the value true) is always associated with a Boolean type variable. The format expression does not apply to declarations of Boolean type quantities.

Only a string format can be directed to string quantities.

Any string value (open quotation) is regarded as having a conditional additional measurement, each position of which corresponds to a specified element of that quotation. The

size of this measurement (the number of elements) is determined by the given format, while for string type variables without a given format it is found, using the SIZE function (see section 3.2.4), according to the current value of the variable.

Format expressions are evaluated once at each entrance into the block. These expressions can depend only on variables and procedures non-local to the block for which the given type declaration is valid. Subscript expressions in repeats are evaluated according to the rules of section 3.1.4.2.

5.1.3.3 A declaration beginning with the declarator as is understood as an extension of a previous variable declaration, the identifier designator of which is placed behind the as declarator, to the variables whose identifiers are carried on the variable identifier list following a semicolon. Thus, if for the following example the variable SHOP has a declaration string SHOP format 'CC', then the cited declaration is understood as the declaration:

string FORM OF PAYMENT format 'CC', OPERATION format 'CC', WAREHOUSE format 'CC'

The semantics of the identifier designator are given in section 5.3.7.2.

5.2 ARRAY DECLARATIONS

5.2.1 Syntax

⟨format array identifier⟩ ::= ⟨array identifier⟩ |
⟨array identifier⟩ format ⟨format expression⟩

⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩ : ⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩ |
    ⟨lower bound⟩ : integer ⟨variable identifier⟩ |
    ⟨bound pair list⟩ , ⟨bound pair⟩
⟨array segment⟩ ::= ⟨format array identifier⟩
    [⟨bound pair list⟩] | ⟨format array identifier⟩ ,
    ⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩ |
    ⟨array list⟩ , ⟨array segment⟩
⟨array identifier designator⟩ ::= ⟨array identifier⟩ |
    ⟨compound identifier⟩ . ⟨array identifier designator⟩
⟨array prototype designator⟩ ::= ⟨array identifier designator⟩
⟨array identifier list⟩ ::= ⟨array identifier⟩ |
    ⟨array identifier list⟩ , ⟨array identifier⟩
⟨array declaration⟩ ::= array ⟨array list⟩ |
    ⟨type⟩ array ⟨array list⟩ | as ⟨array prototype designator⟩
    array ⟨array identifier list⟩

5.2.2 Examples

```
array A, B, C[17 : H, 2 : MB], P[-2 : 10]
integer array A[if C < 0 then 2 else 1 : 20]
real array D[-7 : -1], E format '-T9(9)10-99' [1 : integer C]
Boolean array P, R, S,[1 : integer T, 1 : 2]
string array A format 'C(4)' [1 : 40], B format 'C(K)',
  C, D format 'E', F, G format 'C␣C␣C(T)' [1 : X]
as A arrays B, C, D, E
```

5.2.3 Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts, and the types and possibly the formats of the variables.

5.2.3.1 Subscript Bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript, usually in the form of two arithmetic expressions separated by a semicolon. The bound pair belonging to the high-order measurement (the left-most member of the bound-pair list) can have an upper bound with the following construction:

integer ⟨variable identifier⟩

The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2 Measurements, Dimensions, and Lengths. Each array measurement has a bound pair. The bound pair's sequence number on the bound pair list, counting from left to right, is known as its measurement number. The dimension of an array is defined as the quantity of its measurements. The difference between the value of the upper bound and the value of the lower bound plus 1 is known as the length of each measurement. Array length is understood to be the product of the lengths of all measurements.

5.2.3.3 Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given, the type real is understood.

5.2.3.4 Format Expression. A format expression, when it is explicitly declared, relates to all components of the given array having the identifier associated with the declarator format. The semantics of the format expression are presented in section 5.1.3.2.

5.2.4 Lower and Upper Bound Expressions

5.2.4.1 These expressions are evaluated in the same way as subscript expressions (see section 3.1.4.2).

5.2.4.2 These expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently, in the outermost block of a program only arrays with constant bounds can be declared (see, however, section 5.2.5).

5.2.4.3 Bound expressions should be evaluated once at each entrance into the block.

5.2.5 Special Upper Bound

In the first bound pair, the upper bound can be given in the form of an integer type variable declaration. This means:

5.2.5.1 The identifier of the variable is localized in the same block as the array identifiers connected with the upper bound.

5.2.5.2 The value of the corresponding upper bound is undefined at the time of entrance into the block. It can be given and, if necessary, changed either by an assignment statement or as a result of the execution of an input procedure statement for one of the arrays associated with that

bound. In both cases, the change in the value of the upper bound is extended to all arrays associated with it.

5.2.6 The Identity of Subscripted Variables

The identity of subscripted variables is not related to the subscript bounds given in the array declaration. However, the values of subscripted variables are defined, at any time, only for those variables whose subscripts at the time of the most recent evaluation of variables remained within the given subscript bounds.

Any array is defined only if all its measurement lengths are positive.

5.2.7 Ordering of Subscripted Variables

Array components, even if the array dimension is greater than 1, are ordered linearly according to the lexicographic principle--i.e., component

$$a[k_1,k_2,\ldots,k_m]$$

precedes component

$$a[j_1,j_2,\ldots,j_m]$$

if

$$\left.\begin{array}{l} k_i = j_i, \quad i = 1,2,\ldots,p-1 \\ \\ k_p < j_p \end{array}\right\} \quad 1 \leq p \leq m$$

5.2.8 Declaration According to Prototype

If an array declaration begins with the declarator as, it means that the arrays whose identifiers are enumerated on the list included in that declaration have component values with the same dimension, subscript bounds, type, and format (if such is given for the prototype) as the prototype array whose identifier designator follows the as declarator.

The semantics of the identifier designator are given in section 5.3.7.2.

If an array declaration serving as a prototype contains a special upper bound, then the upper bound of the arrays declared by the prototype are defined according to the current value of the prototype bound. If this value is undefined, then the array declared by the prototype is also undefined.

5.3 COMPOUND DECLARATIONS

5.3.1 Syntax

⟨simple compound identifier list⟩ ::= ⟨simple compound identifier⟩ | ⟨simple compound identifier list⟩ , ⟨simple compound identifier⟩

⟨structure element declaration⟩ ::= ⟨type declaration⟩ | ⟨array declaration⟩ | ⟨compound declaration⟩

⟨structure declaration list⟩ ::= ⟨structure element declaration⟩ | ⟨structure declaration list⟩ ; ⟨structure element declaration⟩

⟨structure declaration⟩ ::= ⟨structure element declaration⟩ | ((⟨structure declaration list⟩))

⟨simple compound segment⟩ ::= ⟨simple compound identifier list⟩ . ⟨structure declaration⟩

⟨simple compound list⟩ ::= ⟨simple compound segment⟩ | ⟨simple compound list⟩ , ⟨simple compound segment⟩

⟨compound-array identifier list⟩ ::= ⟨compound-array identifier⟩ | ⟨compound-array identifier list⟩ , ⟨compound-array identifier⟩

⟨compound-array segment⟩ ::= ⟨compound-array identifier list⟩ [⟨bound pair list⟩] . ⟨structure declaration⟩

⟨compound-array list⟩ ::= ⟨compound-array segment⟩ | ⟨compound-array list⟩ , ⟨compound-array segment⟩

⟨compound identifier⟩ ::= ⟨simple compound identifier⟩ | ⟨compound-array identifier⟩

⟨simple compound identifier designator⟩ ::= ⟨simple compound identifier⟩ | ⟨compound identifier⟩ . ⟨simple compound identifier designator⟩

⟨simple compound prototype⟩ ::= ⟨simple compound identifier designator⟩ | ⟨compound-array identifier designator⟩ [⟨subscript list⟩]

⟨compound-array identifier designator⟩ ::= ⟨compound-array identifier⟩ | ⟨compound identifier⟩ . ⟨compound-array identifier designator⟩

⟨compound-array prototype⟩ ::= ⟨compound-array identifier designator⟩

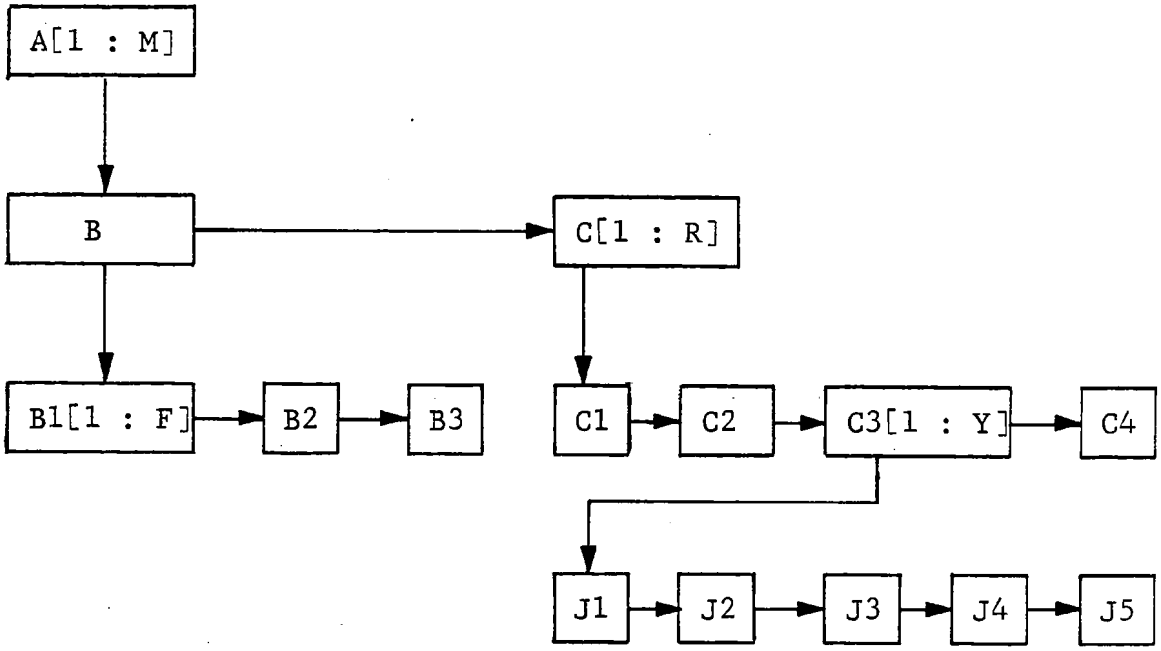
⟨compound declaration⟩ ::= compound ⟨simple compound list⟩ | compound array ⟨compound-array list⟩ | as ⟨simple compound prototype⟩ compound ⟨simple compound identifier list⟩ | as ⟨compound-array prototype⟩ compound array ⟨compound-array identifier list⟩

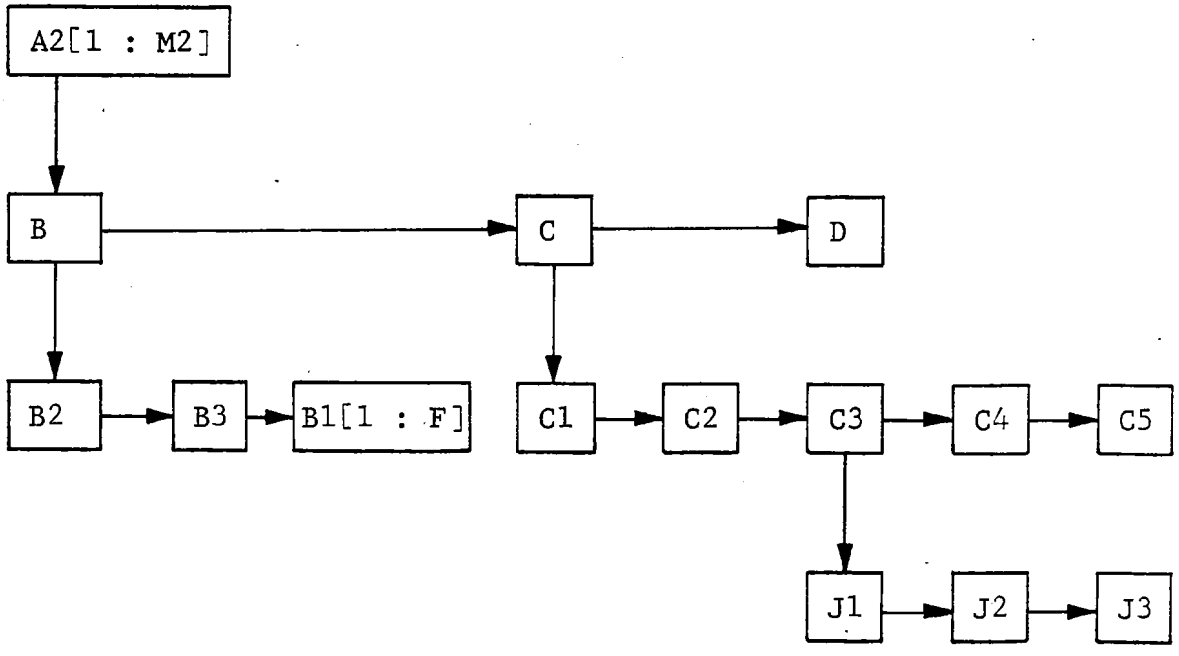
5.3.2 Examples

```
compound arrays A[1 : M].
    (compound B.
        (string array B1[1 : F];
         strings B2, B3);
         compound array C[1 : R].
             (strings C1, C2 format 'S(4)';
              compound array C3[1 : Y].
                  (reals J1, J2, J3, J4, J5);
              integer C4)),
A2[1 : M2].
    compounds B.
        (strings B2, B3 format 'SSS';
         string array B1[1 : F]),
    C.
        (strings C1, C2;
         compound C3.
             (real J1:
              integer J2;
              real J3);
         real C4;
         integer C5);
    integer D);
as A compound arrays D1, D2, D3
```

In the first example, a declaration is made for two compound arrays consisting of similar elements not identically distributed in their structures in some cases and

sometimes with different declarations. For this example, the compounds A and A2 are depicted below schematically.





According to the second example, compounds D1, D2, and D3 have the same element structure and declaration as compound A.

5.3.3 Semantics

A compound declaration introduces into the program a series of identifiers representing one or several compounds and structure elements of such compounds. A compound can be either simple--i.e., consisting of one set of structure elements--or a compound-array, in which case it consists of many such sets, identical in structure. Each such set is considered a compound-array component. Variables--constituents, arrays, and compounds--can be structure elements. A compound declaration assigns a structure, dimension, and subscript bounds to a compound if it is a compound-array, and also declares its structure elements.

5.3.3.1 Levels. A compound that is not a structure element of a different compound is considered a first-level quantity. Structure elements of a K-th level compound are considered to be quantities of the (K + 1)-th level. The higher the level, the greater its number. Variables (constituents, arrays, and compounds) belonging to a level higher than the first cannot be considered separately from the compound of which they are the structure elements.

The level of an identifier declared in a compound matches the level of the quantity designated by that identifier.

5.3.3.2 Lengths. The length of a variable-constituent whose identifier is included in the type declaration in the corresponding structure declaration equals 1. The length of an array that is not a compound-array is defined as in

section 5.2.3.2, regardless of its level. The length of a simple compound, and also of each component of a compound-array, is equal to the sum of the lengths of all elements of its structure. The length of a compound-array is equal to the product of the array length in the sense of section 5.2.3.2 for the length of its components.

5.3.4 Primary Components of a Compound

Variable-constituents (see section 3.1.5) of a given compound that have no format indication are called primary components of that compound. Obviously, the number of different primary components equals the length of the compound. If given, types and formats for primary components are defined individually according to the type declarations and the declarations of highest-level arrays. These types and formats can be different within a single compound.

5.3.5 Structure Element Identifiers

These identifiers must be different within a single structure declaration. However, they can be the same as identifiers of lower-level compounds, even if these compounds are convoluted for the given structure element, and the same as identifiers of composition elements of such compounds.

5.3.6 Ordering of Primary Components

All primary components of compounds are ordered linearly according to the following rules.

5.3.6.1 Compound-array components are ordered depending on the subscript values, in the same way as for array components (see section 5.2.7). If in this sense component A of a compound-array precedes component B of the same compound, then each primary component of A precedes each primary component of B.

5.3.6.2 Simple compound structure elements or structure elements of one component of a compound-array are arranged in the order in which their identifiers occur in the structure declaration. If structure element A precedes structure element B in this sense, then any primary component of element A, or that element itself if it is a variable-constituent, precedes any primary component of element B, or element B itself.

5.3.6.3 The components of any array that is a structure element of a compound are ordered according to the rule of section 5.2.7.

5.3.7 Declaration According to Prototype

5.3.7.1 If a compound declaration begins with the declarator as, it means that the compounds whose identifiers are enumerated in the corresponding identifier list completely duplicate the compound-prototype structure whose identifier designator follows the as declarator. In these compounds in particular, all higher-level compound-prototype identifiers for designating corresponding structure elements are repeated. If a simple compound prototype is given in the form of a compound-array identifier designator with a subscript list, it means that one of the components (any one) of that compound array is a prototype.

5.3.7.2 An identifier designator sequentially arranged according to increasing levels enumerates the compound identifier into which the quantity designated by a finite identifier of that designator enters. A compound prototype designator must begin with a first-level identifier. Higher-level compound identifiers can in some cases be omitted. Specifically, the identifier designator

I.I1.I2.....IK. ID

(where I, I1, I2,...,IK are compound identifiers, and ID is an identifier designator) can be shortened to

I. ID

if any identifier designator

I. ID1

(where I is the same identifier and ID1 is an identifier designator with the same initial identifier as ID) can arise only in abbreviating the full identifier designator

I.I1.I2.....IK. ID1 .

5.3.7.3 If, in the declaration of a compound serving as a prototype, there occurs an array declaration with a special upper bound, then the corresponding arrays, entering the compound declared by that prototype, receive as an upper bound of the first subscript the current values of the upper bound of the first array-prototype subscript. If any of these values is undefined, then the array entering according to the declaration of the prototype is also undefined. Other list structure elements declared according to prototype can be defined thusly.

5.4 SWITCH DECLARATIONS

5.4.1 Syntax

⟨switch list⟩ ::= ⟨label⟩ | ⟨switch list⟩⟨label⟩

⟨switch declaration⟩ ::= switch ⟨switch identifier⟩
:= ⟨switch list⟩

5.4.2 Examples

```
switch S := S1, S2, S3, S4
switch K := R1, L
```

5.4.3 Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one in the form of labels entered in the switch list. With each of these labels there is associated a positive integer, 1, 2,..., obtained by counting the items in the list from left to right. A label with a subscript expression value given by its sequence number on the switch list is the value of the switch designator corresponding to the given value of the subscript expression (see section 3.6, Designational Expressions).

5.4.4 Influence of Scopes

If a switch designator occurs outside the scope of a label entering into a switch list, and an evaluation of this switch designator selects this precise label, then the conflicts between the identifier designating it and the identifiers whose declarations are valid at the place of

the switch designator are avoided through suitable systematic changes of the latter identifiers.

5.5 PROCEDURE DECLARATIONS

5.5.1 Syntax

⟨formal parameter⟩ ::= ⟨identifier⟩

⟨formal parameter list⟩ ::= ⟨formal parameter⟩ |
⟨formal parameter list⟩⟨parameter delimiter⟩
⟨formal parameter⟩

⟨formal parameter part⟩ ::= (⟨formal parameter list⟩) | ⟨empty⟩

⟨identifier list⟩ ::= ⟨identifier⟩ | ⟨identifier list⟩ ,
⟨identifier⟩

⟨value part⟩ ::= value ⟨identifier list⟩ ; | ⟨empty⟩

⟨specificator⟩ ::= ⟨type⟩ | array | ⟨type⟩ array | compound |
compound array | label | switch | procedure | ⟨type⟩
procedure

⟨specifier⟩ ::= ⟨specificator⟩⟨identifier list⟩ ;

⟨specification part⟩ ::= ⟨empty⟩ | ⟨specifier⟩ |
⟨specification part⟩⟨specifier⟩

⟨procedure heading⟩ ::= ⟨procedure identifier⟩
⟨formal parameter part⟩⟨value part⟩⟨specification part⟩

⟨procedure body⟩ ::= ⟨statement⟩

⟨procedure declaration⟩ ::= procedure ⟨procedure heading⟩
⟨procedure body⟩ | ⟨type⟩ procedure ⟨procedure heading⟩
⟨procedure body⟩

5.5.2 Examples (see also the examples at the end of the report)

```
procedure SPUR (A) ORDER: (N) RESULT TO: (S); value N;
  array A; integer N; real S;
begin integer K;
  S := 0;
  for K := 1 : N do S := S + A[K, K]
end

procedure TRANSPOSE (A) ORDER: (N); value N;
  array A; integer N;
begin real W; integers I, K;
  for I := 1 : N do
  for K := 1 + I : N do
  begin W := A[I, K];
    A[I, K] := A[K, I];
    A[K, I] := W
  end
end TRANSPOSE

integer procedure STEP (U); real U;
STEP := if 0 ≤ U ∧ U ≤ 1 then 1 else 0

procedure ABSMAX (A) SIZES: (N, M) RESULT TO: (Y)
  SUBSCRIPTS: (I, K);
comment The absolute greatest element of the matrix
  A, of size N by M is transferred to Y, and the
  subscripts of this element to I and K;
  array A; integers N, M, I, K; real Y;
begin integers P, Q; Y := -1;
  for P := 1 : N do
  for Q := 1 : M do
```

```
  if ABS(A[P, Q]) > Y then
  begin Y := ABS(A[P, Q]);
    I := P; K := Q
  end
end ABSMAX

procedure INNERPRODUCT (A, B) ORDER: (K, P) RESULT TO: (Y);
  value K; integers K, P; reals Y, A, B;
begin real C; C := 0;
  for P := 1 : K do C := C + A × B;
  Y := C
end INNERPRODUCT

 procedure SELECT (A, NA, B, NB, I, L); value NA;
   compounds A, B; integers NA, NB, I; Boolean L;
 begin NB := 0;
   for I := 1 : NA do if L then
   begin NB : NB + 1;
     B := A
   end
 end SELECT

 integer procedure NUMBER OF SYMBOLS (S) IN TEXT: (T);
   value S; strings S, T;
 begin integers D, I, K;
   D := SIZE (T); I := 0;
   for K := 1 step 1 until D do
   if T (element K) = S then I := I + 1;
   NUMBER OF SYMBOLS := I
 end
```

5.5.3 Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement, the procedure body, which through the use of a procedure statement or a function designator can be activated from other parts of the block in whose head the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body should, whenever the procedure is activated (see section 3.2, Function Designators, and section 4.7, Procedure Statements), be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal should be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently, the scope of any label labeling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label, as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity. No identifier can occur more than once in a formal parameter list.

5.5.4 Values of Function Designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. The type associated with the procedure identifier must be declared through the inclusion of a type declarator as the very first symbol of the procedure declaration.

A function designator and the procedure declaration defining its value must be such that any possible use of the function designator as a procedure statement will be equivalent to an empty statement.

5.5.5 Specifications

A specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, can be included in a heading. In this part, each formal parameter, if the procedure has any, can occur once.

EXAMPLES

Example 1

PROGRAM 12345600:

begin

Comment. Declarations are submitted in the following format for ease of visualization:

Declarations	Identifiers	format	Format	Bound Pairs
compound arrays	WORK-CARD			[1 : integer T].
	NEW-WORK-CARD			
strings	NAME	format	'S(15)',	
	DEPT	format	'S(5)';	
array	TIME	format	'99T99'	[1 : 5];
reals	TOTAL-TIME	format	'99T99',	
	AVERAGE-TIME	format	'99T99';	
real	V;			
integers	H, N;			

Comment. At this point in the program there must be a punch-card input statement for the WORK-CARD compound-array;

```
for N := 1 : T do
begin V := 0;
  for N := 1 : 5 do
  V := V + WORK-CARD [H]. TIME [N];
  WORK-CARD [H]. TOTAL-TIME := V;
  WORK-CARD [H]. AVERAGE-TIME := V/5;
  NEW-WORK-CARD [H] := WORK-CARD [H];
```

Comment. At this point in the program there must be a statement for outputting the values of NEW-WORK-CARD [H] to the printer;

end DO H.

end PROGRAM 12345600.

For comparison purposes, this same program is presented [on the following page] written in COBOL.*

Example 2

Problem Statement:

The initial data are represented on a Work-Card in the following manner:

WORK-CARD [1 : T]						
Dept	Employee Work-Card Number	Time Worked (per day)				
		1	2	3	...	31

Background Data, given in the following form, are needed for accounting purposes:

BACKGROUND DATA [1 : S]					
Dept	Employee Work-Card Number	Last Name, First, Middle	Job	Employee Category	Hourly Rate

*Ref. 7, pp. 151-153.

```
010010 IDENTIFICATION DIVISION.
   020 PROGRAM-ID. 12345600.
   030 ENVIRONMENT DIVISION.
   040 CONFIGURATION SECTION.
   050 SOURCE-COMPUTER.
   060 OBJECT-COMPUTER.
   070 INPUT-OUTPUT SECTION.
   080 FILE-CONTROL.
   090 I-O-CONTROL.
   100 DATA DIVISION.
   110 FILE SECTION.
   120 FD  WORK-FILE
   130     LABEL RECORDS ARE OMITTED
   140     DATA RECORD IS WORK-CARD.
   150 01 WORK-CARD
   160     02 NAME PICTURE X(15).
   170     02 DEPT PICTURE X(5).
   180     02 TIME PICTURE 99V99 OCCURS 5 TIMES.
   190     02 TOTAL-TIME PICTURE 99V99.
   200     02 AVERAGE-TIME PICTURE 99V99.
   210     02 FILLER PICTURE X(32).
   220 FD NEW-WORK-FILE LABEL RECORDS ARE OMITTED DATA RECORD IS NEW-
   230-   WORK-CARD.
   240 01  NEW-WORK-CARD PICTURE IS X(80).
   250 WORKING-STORAGE SECTION.
020010 77  N SIZE IS 1 COMPUTATIONAL NUMERIC DIGIT.
   020 PROCEDURE DIVISION.
   030 1. OPEN INPUT WORK-FILE OUTPUT NEW-WORK-FILE.
   040 2. READ WORK-FILE RECORD AT END GO TO 3.
   050     MOVE ZEROS TO TOTAL-TIME   PERFORM 4 VARYING N FROM 1 BY 1 UN
   060-    TIL N GREATER 5  DIVIDE 5 INTO TOTAL-TIME GIVING AVERAGE-TIME
   070     ROUNDED   WRITE NEW-WORK-CARD FROM WORK-CARD GO TO 2.
   080 3.  CLOSE WORK-FILE AND NEW-WORK-FILE                    STOP RUN.
   090 4.  ADD TIME (N) TO TOTAL-TIME.
```

Program of Example 1 Written in COBOL

It is required that a Payroll Account by employee per pay period be maintained, with the following format:

ACCOUNT [1 : T]						
Dept	Name	Employee Work-Card Number	Time (per month)		Wages	Remarks
			Hrs	Days		
TOTAL:						

The names of employees are entered to the Account from the Background Data; they are entered there at the same time as Work-Card Numbers (together with Department Numbers) are copied from the Work-Cards to the Account. Only the last name and first letters of the first and middle names (the initials) are entered to the Account (all names are given in full in the Background Data).*

Monthly time in hours is determined for the Account by totaling the data on the Work-Card, while monthly time in days is the number of days on the Work-Card for which hour entries have been made.

The total wage charge on the Account is determined by multiplying the employee's time worked during the month in hours by his hourly rate (obtained from the Background Data).

---

*Subsequently in this example, "Name" means last name and first initials; "Full Name" means last, first, and middle names.--Trans.

In the event of absence of Background Data on an employee, his Work-Card Number and department number are placed on a list of nonformulated employees, and a notation is made in the Account to this effect. Wages for any such employee are not computed.

Column totals are produced on the Account for Hours, Days, and Wages.

Based on these data, it is also required to produce a resume of the Distribution in Man-Days of employee time by Job and Category:

Job	Distribution in Man-Days (%)					
	by Category [man-days]					by Job [%]
	1	2	3	4	5	
Assembly worker, etc.	15.0	25.0	30.0	10.0	20.0	5.0
TOTAL [%]:	10.0	20.0	30.0	35.0	5.0	100

In distributing man-days according to Category, the total hours expended in man-days (the "days" column on the Account) for the given Job is used. The total by Job is based in the Distribution on percentage. These time totals are determined from data in the Account (Work-Card Number and Days columns) and in the Background Data (Work-Card Number, Job, and Employee Category columns).

In distributing man-days by Job (last column of the resume), the 100% figure is of the overall time totals in man-days.

Percentage data in the resume must be given in the alphabetical order of Job titles.

Additionally, it is required that a new Work-Card be prepared for subsequent recording of employee hours. The form of the new Work-Card, in which only the first two columns are filled in, is the same as the one shown above.

Data editing requirements are indicated in their declarations in the program.

It is assumed that there is a SORT procedure declaration in the library that orders the values of a one-dimensional string array lexicographically. The form of the statement for this procedure is as follows:

LIBRARY ('SORT', ⟨array identifier⟩, ⟨lower bound⟩,
⟨upper bound⟩)

It is also assumed that the variables S and P and procedures with identifiers SEARCH, MULTIPLY, and WAGE are declared for the external block. Declarations for these procedures are presented below.

```
procedure SEARCH (I, N, V, L1, L2) values N, V;
  integers I, N, V; Booleans L1, L2;
begin L2 := true;
  for I := N : V do
  if L1 then
  begin L2 := false;
    to M
  end;
  M:
end
```

```
string procedure MULTIPLY (T, A); values T, A;
  integer A; string T;
begin integer I; string X;
  X := '';
  for I := 1 : A do
  X := X ← T;
  MULTIPLY := X
end
```

```
real procedure WAGE (M, I1, I2); values I1, I2;
  integers I1, I2; array M;
begin integer I; WAGE := 0;
  for I := I1 : I2 do
  WAGE := WAGE + M[I]
end
```

Problem Solution:

PAYROLL OVER TIME:

```
begin
  Comment. Declarations are submitted in the following
    form for ease of visualization:
```

Declarations	Identifiers	format	Format	Bound Pairs
compound arrays	BACKGROUND			[1 : S].
(strings	DEPT	format	'SS',	
	W-C NO	format	'S(4)',	
	FULL NAME	format	'S(70)',	
	JOB	format	'S(30)',	
	CATEGORY	format	'S';	
real	RATE	format	'9.99'),	

	WORK-CARD			[1 : integer T].
(strings	DEPT	format	'SS',	
	W-C NO	format	'SSSS';	
real array	HOURS	format	'99T99'	[1 : 31]),
	ACCOUNT			[1 : integer T1].
(strings	DEPT	format	'SS',	
	NAME	format	'S(45)',	
	W-C NO	format	'S(4)';	
real	HOURS	format	'PPP.9U';	
integer	DAYS	format	'PP';	
real	WAGE	format	'PPPT99';	
string	REMARKS	format	'S(30)'),	
	SV			[1 : P].
(string	JOB	format	'S(30)';	
integer array	CATEGORY	format	'P(4)'	[1 : 5];
integer	TOTAL	format	'P(6)'),	
	RESUME			[1 : P + 1].
(string	JOB	format	'S(30)';	
compound	DISTRIBUTION IN MAN DAYS.			
(string array	FOR CATEGORIES			[1 : 5];
string	FOR JOBS)),			
	NONFORMULATED			[1 : integer N].
(strings	DEPT	format	'SS',	
	W-C NO	format	'S(4)');	

compounds (real integer real	TOTALS V. HOURS DAYS WAGE	 format format format	 'P(7).9'; 'P(6)'; '*(6).99'),	
(integer array integer	TOTALS SV. CATEGORY TOTAL	 format format	 'P(6)' 'P(6)');	 [1 : 5];
string array	PROF	format	'S(30)'	[1 : P];

```
integers IT, IS, IR, IC, IP, I, R; Booleans ABSENT;
strings BLANKS, F;

Preparation:
  Comment. At this point in the program there must be input
    statements for the BACKGROUND and WORK-CARD compound arrays;
  ACCOUNT. DAYS := 0; ACCOUNT. WAGE := 0;
  BLANKS := MULTIPLY ('␣', 30);
  ACCOUNT. REMARKS := ''; ACCOUNT. NAME := '';
  SV. JOB := '';
  SV. (CATEGORY, TOTAL) := 0;
  TOTALS SV := 0;
  TOTALS V. (HOURS, WAGE) := 0; TOTALS V. DAYS := 0;
  N := 0;
  F := 'S(5)'␣'%'';
  begin as WORK-CARD compound array NEW-WORK-CARD;
    NEW-WORK-CARD. HOURS := 0;
    TI := T;
    for IT := 1 : T do
```

```
begin ACCOUNT [IT]. (DEPT, W-C NO) :=
  WORK-CARD [IT]. (DEPT, W-C NO);
  NEW-WORK-CARD [IT]. (DEPT, W-C NO) :=
    WORK-CARD [IT] . (DEPT, W-C NO);
  ACCOUNT [IT]. HOURS := WAGE (WORK-CARD [IT].
    HOURS, 1, 31);
  for IC := 1 :: 31 do
  ACCOUNT [IT]. DAYS := ACCOUNT [IT]. DAYS +
    (if WORK-CARD [IT]. HOURS [IC] > 0 then 1 else 0);
  SEARCH (IS, 1, S, BACKGROUND [IS].
    W-C NO = ACCOUNT [IT]. W-C NO, ABSENT);
  A : if ABSENT then
  begin N := N + 1;
    NONFORMULATED [N] := WORK-CARD [IT]. (DEPT, W-C NO);
    ACCOUNT [IT]. REMARKS := 'NO␣BACKGROUND␣DATA'
  end
  else
  begin FAM:
    procedure DO (M1, MO); labels M1, MO;
    begin IR := IR + 1;
      if IR > 70 then to M9;
      if BACKGROUND [IS]. FULL NAME [element IR] = '␣'
      then to M1 else to MO
    end;
    FAM := '';
    IR := 0;
    M1 : DO (M1, M2);
    M2 : FAM := FAM ← BACKGROUND [IS].
      FULL NAME [element IR]; DO (M3, M2);
    M3 : FAM := FAM ← '␣';
    M4 : DO (M4, M5);
    M5 : FAM := FAM ← BACKGROUND [IS].
      FULL NAME [element IR] ← '.␣';
```

```
      M6 : DO (M7, M6);
      M7 : DO (M7, M8);
      M8 : FAM := FAM ← BACKGROUND [IS].
        FULL NAME [element IR] ← '.';
      M9 : ACCOUNT [IT]. NAME := FAM;
      ACCOUNT [IT] . WAGE := ACCOUNT [IT].
        HOURS × BACKGROUND [IS]. RATE;
      SEARCH (IP, 1, P, SV [IP]. JOB = BACKGROUND [IS].
        JOB, ABSENT);
      if ABSENT then
      begin SEARCH (IP, 1, P, SV [IP].
        JOB = BLANKS, ABSENT);
        if ABSENT then to ERROR 1 else
        begin SV [IP]. JOB := BACKGROUND [IS]. JOB;
          SV [IP]. CATEGORY [BACKGROUND [IS].
            CATEGORY sense '9'] := ACCOUNT [IT]. DAYS
        end
      end
      else SV [IP]. CATEGORY [BACKGROUND [IS].
        CATEGORY sense '9'] :=
      SV [IP]. CATEGORY [BACKGROUND [IS].
        CATEGORY sense '9'] + ACCOUNT [IT]. DAYS
    end A;
    TOTALS V. HOURS := TOTALS V. HOURS + ACCOUNT [IT].
      HOURS;
    TOTALS V. DAYS := TOTALS V. DAYS + ACCOUNT [IT].
      DAYS;
    TOTALS V. WAGE := TOTALS V. WAGE + ACCOUNT [IT].
      WAGE
  end IT.
```

```
    Comment. ACCOUNT, TOTALS V, NEW-WORK-CARD, and
      NONFORMULATED are obtained and output to the printer;
  end NEW-WORK-CARD;
  for R := 1 : 5 do
  begin SV. TOTAL := SV. TOTAL + SV. CATEGORY [R];
    for IP := 1 : P do
    TOTALS SV. CATEGORY [R] :=
    TOTALS SV. CATEGORY [R] + SV [IP]. CATEGORY [R];
    TOTALS SV. TOTAL := TOTALS SV. TOTAL + TOTALS SV.
      CATEGORY [R]
  end
  PROF := SV. JOB;
  LIBRARY ('SORT', PROF, 1, P);
  BACKGROUND. JOB := PROF;
  for I := 1 : P do
  begin SEARCH (IP, 1, P, BACKGROUND [I]. JOB = SV [IP].
    JOB, ABSENT);
    BACKGROUND [I]. FOR CATEGORIES := (SV [IP].
      CATEGORY x 100/SV [IP]. TOTAL) text 'PPP.9' text F;
    BACKGROUND [I]. FOR JOBS := (SV [IP].
      TOTAL x 100/TOTALS SV. TOTAL) text 'PPP.9' text F
  end I;
  BACKGROUND [P + 1]. FOR CATEGORIES := (TOTALS SV.
    CATEGORY x 100/TOTALS SV. TOTAL) text 'PPP.9' text F;
  BACKGROUND [P + 1]. FOR JOBS := '100.0' text F;
  BACKGROUND [P + 1]. JOB := '␣␣␣␣␣TOTAL :';
  Comment. At this point the BACKGROUND compound array
    must be output to the printer;
end PROGRAM.
```

REFERENCES

1. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, "Revised Report on the Algorithmic Language ALGOL 60," Peter Naur (ed.), IFIP, 1962 [Commun. ACM, Vol. 6, No. 1, January 1963, pp. 1-17; Comp. J., Vol. 5, No. 4, January 1963, pp. 349-367; Numer. Math., Vol. 4, No. 5, 1963, pp. 420-453].

2. "Report on SUBSET ALGOL 60 (IFIP)," IFIP Working Group 2.1 on ALGOL, Princeton, New Jersey, 1964 [Commun. ACM, Vol. 7, No. 10, October 1964, pp. 626-628].

3. "Report on Input-Output Procedures for ALGOL 60," IFIP Working Group 2.1 on ALGOL, Princeton, New Jersey, 1964.

4. Report to Conference on Data Systems Languages Including Revised Specifications for a Common Business Oriented Language (COBOL) for Programming Electronic Digital Computers, Department of Defense, 1961.

5. D. E. Knuth, L. L. Bumgarner, P. Z. Ingerman, J. H. Merner, D. E. Hamilton, M. P. Lietzke, and D. T. Ross, "A Proposal for Input-Output Conventions in ALGOL 60 (A Report of the Subcommittee on ALGOL of the ACM Programming Languages Committee)," Commun. ACM, Vol. 7, No. 5, May 1964 [pp. 273-283].

6. J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," ICIP, Paris, June 1959.

7. IBM, General Information Manual COBOL, Form F28-8053-1.

RUSSIAN-ENGLISH GLOSSARY

¯,%,◇,|,_,–,!,≫,°,´,¬,?,↓,Ø,±,∇ , см. ⟨специальный знак⟩
„“ , см. кавычки для строк
+ , см. плюс
– , см. минус
× , см. умножение
/,÷ , см. деление
↑, см. возведение в степень
<,≤,≐,≥,>,≠ , см. ⟨знак операции отношения⟩
≡,⊃,∨,∧,¬ , см. ⟨знак логической операции⟩
← , см. операция присоединения
, , см. запятая
. , см. десятичная точка
10 , см. десять
: , см. двоеточие
; , см. точка с запятой
␣ , см. пробел
* , см. звездочка
( ) , см. скобки
[ ] , см. индексные скобки
:= , см. двоеточие равенство

⟨+ позиции⟩ ⟨+ positions⟩
⟨+ часть⟩ ⟨+ part⟩
⟨– позиции⟩ ⟨- positions⟩
⟨– часть⟩ ⟨- part⟩
⟨* позиции⟩ ⟨* positions⟩
⟨* часть⟩ ⟨* part⟩
⟨9 позиции⟩ ⟨9 positions⟩
⟨9 часть⟩ ⟨9 part⟩

⟨арифметическое выражение⟩ ⟨arithmetic expression⟩
⟨арифметическое отношение⟩ ⟨arithmetic relation⟩

⟨безусловный оператор⟩ ⟨unconditional statement⟩
⟨блок⟩ ⟨block⟩
⟨буква⟩ ⟨letter⟩

⟨верхняя граница⟩ ⟨upper bound⟩
вещественная см. вещественное
вещественное real
вещественные см. вещественное
вещественный см. вещественное
вид format
возведение в степень ↑ exponentiation
⟨вставка⟩ ⟨insert⟩
⟨вторичное логическое выражение⟩ ⟨Boolean secondary⟩
⟨вторичное многокомпонентное выражение⟩ ⟨secondary multicomponent expression

⟨вторичное текстовое выражение⟩ ⟨secondary string expression⟩
⟨выражение⟩ ⟨expression⟩

⟨граничная пара⟩ ⟨bound pair⟩

двоеточие : colon
двоеточие равенство := colon equal
деление /,÷ divide
десятичная точка . decimal point
⟨десятичное число⟩ ⟨decimal number⟩
десять $_{10}$ ten
для for
до until

если if

⟨заголовок процедуры⟩ ⟨procedure heading⟩
⟨заголовок цикла⟩ ⟨for clause⟩
запятая , comma
звездочка * asterisk
⟨знак арифметической операции⟩ ⟨arithmetic operator⟩
⟨знак логической операции⟩ ⟨logical operator⟩
⟨знак мантиссы⟩ ⟨mantissa sign⟩
⟨знак операции⟩ ⟨operator⟩
⟨знак операции отношения⟩ ⟨relational operator⟩
⟨знак операции следования⟩ ⟨sequential operator⟩
⟨знак операции типа сложения⟩ ⟨adding operator⟩
⟨знак операции типа умножения⟩ ⟨multiplying operator⟩
⟨знак текстовой операции⟩ ⟨string operator⟩
⟨знаковая часть⟩ ⟨sign part⟩
значение value

⟨идентификатор⟩ ⟨identifier⟩
⟨идентификатор массива⟩ ⟨array identifier⟩
⟨идентификатор массива с форматом⟩ ⟨format array identifier⟩
⟨идентификатор переключателя⟩ ⟨switch identifier⟩
⟨идентификатор переменной⟩ ⟨variable identifier⟩
⟨идентификатор простой составной⟩ ⟨simple compound identifier⟩
⟨идентификатор процедуры⟩ ⟨procedure identifier⟩
⟨идентификатор составной⟩ ⟨compound identifier⟩
⟨идентификатор составной-массива⟩ ⟨compound-array identifier⟩
⟨именующее выражение⟩ ⟨designational expression⟩
⟨имплпкация⟩ ⟨implication⟩
иначе else
⟨индексное выражение⟩ ⟨subscript expression⟩
индексные скобки [] subscript brackets
истина true

кавычки для строк '' quotation marks
как as
конец end
⟨конец переменной-составляющей⟩ ⟨variable-constituent tail⟩
⟨конец составного оператора⟩ ⟨compound statement tail⟩
⟨левая часть⟩ ⟨left part⟩
логическая см. логическое
логические см. логическое
логический см. логическое
⟨логический одночлен⟩ ⟨Boolean monomial⟩
⟨логический терм⟩ ⟨Boolean term⟩
логическое Boolean
⟨логическое выражение⟩ ⟨Boolean expression⟩
⟨логическое значение⟩ ⟨logical value⟩
ложь false

массив array
⟨масштаб⟩ ⟨scale⟩
⟨масштабированный формат правильной дроби⟩ ⟨proper fraction scaled format⟩
⟨метка⟩ ⟨label⟩
метка label
минус - minus
⟨многокомпонентная левая часть⟩ ⟨multicomponent left part⟩
⟨многокомпонентное выражение⟩ ⟨multicomponent expression⟩
⟨многокомпонентное условие⟩ ⟨multicomponent if clause⟩
⟨множитель⟩ ⟨factor⟩

на to
⟨наименование составной⟩ ⟨compound name⟩
начало begin
⟨начало блока⟩ ⟨block head⟩
⟨неименующее выражение⟩ ⟨nondesignational expression⟩
⟨непомеченный блок⟩ ⟨unlabeled block⟩
⟨непомеченный основной оператор⟩ ⟨unlabeled basic statement⟩
⟨непомеченный составной оператор⟩ ⟨unlabeled compound statement⟩
⟨нижняя граница⟩ ⟨lower bound⟩

⟨ограничитель⟩ ⟨delimiter⟩
⟨ограничитель параметра⟩ ⟨parameter delimiter⟩
⟨оператор⟩ ⟨statement⟩
⟨оператор «если»⟩ ⟨if statement⟩
⟨оператор перехода⟩ ⟨go to statement⟩
⟨оператор присваивания⟩ ⟨assignment statement⟩
⟨оператор процедуры⟩ ⟨procedure statement⟩
⟨оператор цикла⟩ ⟨for statement⟩
операция присоединения ← concatenation operator
⟨описание⟩ ⟨declaration⟩

⟨описание массивов⟩ ⟨array declaration⟩
⟨описание переключателя⟩ ⟨switch declaration⟩
⟨описание процедуры⟩ ⟨procedure declaration⟩
⟨описание составных⟩ ⟨compound declaration⟩
⟨описание структуры⟩ ⟨structure declaration⟩
⟨описание типа⟩ ⟨type declaration⟩
⟨описание элемента структуры⟩ ⟨structure element declaration⟩
⟨описатель⟩ ⟨declarator⟩
⟨основной оператор⟩ ⟨basic statement⟩
⟨основной символ⟩ ⟨basic symbol⟩
⟨открытая строка⟩ ⟨open quotation⟩
⟨отношение⟩ ⟨relation⟩

⟨П позиции⟩ ⟨P positions⟩
⟨П часть⟩ ⟨P part⟩
⟨первичное арифметическое выражение⟩ ⟨primary arithmetic expression⟩
⟨первичное логическое выражение⟩ ⟨Boolean primary⟩
⟨первичное многокомпонентное выражение⟩ ⟨primary multicomponent expression⟩
⟨первичное текстовое выражение⟩ ⟨primary string expression⟩
переключатель switch
плюс + plus
пробел ␣ space
⟨прототип простой составной⟩ ⟨simple compound prototype⟩
⟨прототип составной-массива⟩ ⟨compound-array prototype⟩
процедура procedure
⟨пусто⟩ ⟨empty⟩
⟨пустой оператор⟩ ⟨dummy statement⟩
⟨переключательный список⟩ ⟨switch list⟩
⟨переменная⟩ ⟨variable⟩
⟨переменная с индексами⟩ ⟨subscripted variable⟩
⟨переменная-составляющая⟩ ⟨variable-constituent⟩
⟨перечень шкалы индекса⟩ ⟨subscript scale catalog⟩
⟨повторитель⟩ ⟨repeat⟩
⟨подавляемая часть⟩ ⟨suppressed part⟩
пока while
⟨порядок⟩ ⟨exponent⟩
⟨правильная дробь⟩ ⟨proper fraction⟩
примечание comment
⟨примитивное текстовое выражение⟩ ⟨primitive string expression⟩
⟨программа⟩ ⟨program⟩
⟨простая переменная⟩ ⟨simple variable⟩
⟨простое арифметическое выражение⟩ ⟨simple arithmetic expression⟩
⟨простое логическое выражение⟩ ⟨simple Boolean⟩
⟨простое многокомпонентное выражение⟩ ⟨simple multicomponent expression⟩
⟨простое текстовое выражение⟩ ⟨simple string expression⟩

⟨разделитель⟩ ⟨separator⟩
⟨ряд М⟩ ⟨M series⟩
⟨ряд шпаций⟩ ⟨space series⟩

⟨С позиции⟩ ⟨S positions⟩
⟨сегмент массивов⟩ ⟨array segment⟩
⟨сегмент простых составных⟩ ⟨simple compound segment⟩
⟨сегмент составных-массивов⟩ ⟨compound-array segment⟩
⟨скобка⟩ ⟨bracket⟩
скобки () parentheses
смысл sense
⟨совокупность спецификаций⟩ ⟨specification part⟩
⟨совокупность фактических параметров⟩ ⟨actual parameter part⟩
⟨совокупность формальных параметров⟩ ⟨formal parameter part⟩
⟨состав⟩ ⟨composition⟩
составная см. составное
составное compound
составной см. составное
⟨составной оператор⟩ ⟨compound statement⟩
составные см. составное
⟨специальный знак⟩ ⟨special character⟩
⟨спецификатор⟩ ⟨specificator⟩
⟨спецификация⟩ ⟨specifier⟩
⟨список граничных пар⟩ ⟨bound pair list⟩
⟨список значений⟩ ⟨value part⟩
⟨список идентификаторов⟩ ⟨identifier list⟩
⟨список идентификаторов массивов⟩ ⟨array identifier list⟩
⟨список идентификаторов переменных⟩ ⟨variable identifier list⟩
⟨список идентификаторов простых составных⟩ ⟨simple compound identifier list⟩
⟨список идентификаторов составных-массивов⟩ ⟨compound-array identifier list⟩
⟨список индексов⟩ ⟨subscript list⟩
⟨список левой части⟩ ⟨left part list⟩
⟨список массивов⟩ ⟨array list⟩
⟨список описания структуры⟩ ⟨structure declaration list⟩
⟨список позиций⟩ ⟨position list⟩
⟨список простых составных⟩ ⟨simple compound list⟩
⟨список состава⟩ ⟨composition list⟩
⟨список составных-массивов⟩ ⟨compound-array list⟩
⟨список типа⟩ ⟨type list⟩
⟨список фактических параметров⟩ ⟨actual parameter list⟩
⟨список формальных параметров⟩ ⟨formal parameter list⟩
⟨список цикла⟩ ⟨for list⟩
⟨список шкал индексов⟩ ⟨subscript scale list⟩
⟨строка⟩ ⟨quotation⟩
строка quotation
⟨строка букв⟩ ⟨letter quotation⟩

⟨строчная кавычка⟩ ⟨quotation mark⟩
⟨строчный символ⟩ ⟨quotation symbol⟩

текст **text**
текстовая см. текстовое
текстовое **string**
⟨текстовое выражение⟩ ⟨string expression⟩
⟨текстовое отношение⟩ ⟨string relation⟩
текстовой см. текстовое
⟨текстовой формат⟩ ⟨string format⟩
текстовые см. текстовое
⟨тело процедуры⟩ ⟨procedure body⟩
⟨терм⟩ ⟨term⟩
⟨тип⟩ ⟨type⟩
то **then**
точка с запятой ; **semicolon**

⟨указатель идентификатора массива⟩ ⟨array identifier designator⟩
⟨указатель идентификатора переменной⟩ ⟨variable identifier designator⟩
⟨указатель идентификатора простой составной⟩ ⟨simple compound identifier designator⟩
⟨указатель идентификатора составной-массива⟩ ⟨compound-array identifier designator⟩
⟨указатель масштаба⟩ ⟨scale designator⟩
⟨указатель округления⟩ ⟨roundoff designator⟩
⟨указатель переключателя⟩ ⟨switch designator⟩
⟨указатель подмассива⟩ ⟨subarray designator⟩
⟨указатель прототипа массива⟩ ⟨array prototype designator⟩
⟨указатель прототипа переменной⟩ ⟨variable prototype designator⟩
⟨указатель составляющей⟩ ⟨constituent designator⟩
⟨указатель составной⟩ ⟨compound designator⟩
⟨указатель точки⟩ ⟨point designator⟩
⟨указатель функции⟩ ⟨function designator⟩
умножение × **multiply**
⟨условие⟩ ⟨if clause⟩
⟨условный оператор⟩ ⟨conditional statement⟩

⟨фактический параметр⟩ ⟨actual parameter⟩
⟨формальный параметр⟩ ⟨formal parameter⟩
⟨формат⟩ ⟨format⟩
⟨формат вещественного⟩ ⟨real format⟩
⟨формат десятичного числа⟩ ⟨decimal number format⟩
⟨формат мантиссы⟩ ⟨mantissa format⟩
⟨формат мантиссы со знаком⟩ ⟨signed mantissa format⟩
⟨формат порядка⟩ ⟨exponent format⟩
⟨формат правильной дроби⟩ ⟨proper fraction format⟩
⟨формат целого⟩ ⟨integer format⟩
⟨формат целого без знака⟩ ⟨unsigned integer format⟩
⟨форматное выражение⟩ ⟨format expression⟩

целая см. целое
⟨целая часть⟩ ⟨integer part⟩
⟨целое⟩ ⟨integer⟩
целое integer
⟨целое без знака⟩ ⟨unsigned integer⟩
целые см. целое
целый см. целое
цикл do
⟨цифра⟩ ⟨digit⟩

⟨число⟩ ⟨number⟩
⟨число без знака⟩ ⟨unsigned number⟩
⟨числовой формат⟩ ⟨number format⟩
⟨чистая строка⟩ ⟨blank quotation⟩

шаг step
⟨шкала индекса⟩ ⟨subscript scale⟩
⟨шпация⟩ ⟨space⟩

⟨Э позиции⟩ ⟨E positions⟩
элемент element
⟨элемент состава⟩ ⟨composition element⟩
⟨элемент списка позиций⟩ ⟨position list element⟩
⟨элемент списка типа⟩ ⟨type list element⟩
⟨элемент списка цикла⟩ ⟨for list element⟩
⟨элемент строки⟩ ⟨quotation element⟩
⟨элемент шкалы индекса⟩ ⟨subscript scale element⟩

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

(Including the English-Russian Glossary)

The Index, not included in the Kibernetika article, is patterned after the Index to the "Revised ALGOL 60 Report," adjusted for ALGEC by the translator. The Index does not, however, provide references to text definitions and discussions. Following each entry is the original Russian term.

Index references are to section numbers, and are given in two groups:

def	References following the abbreviation "def" are to the syntactic definition (if any);
synt	References following the abbreviation "synt" are to occurrences in metalinguistic formulae (section numbers in the "def" group are not repeated in the "synt" group).

Basic symbols represented by signs other than underscored words have been collected at the beginning.

The Example sections were ignored in compiling the Index.

BIBLIOGRAPHY OF RAND CORPORATION PUBLICATIONS IN SOVIET CYBERNETICS AND COMPUTER TECHNOLOGY

1. Ware, W. H. (ed.), Soviet Computer Technology--1959 RM-2541, March 1, 1960. Reprinted in IRE Transactions on Electronic Computers, Vol. EC-9, No. 1, March 1960.

   An account of a trip taken by two RAND computer specialists to the Soviet Union as part of an eight-man delegation representing the U.S. National Joint Computer Committee and its member societies. The genesis of the delegation and its itinerary in the Soviet Union are traced. The state of the art in Soviet computer technology as observed by the delegates is examined, showing the development, constructions, applications, routines, and components of the major Soviet computing machines. Impressions are included on Soviet education, the role of the Academy of Sciences, and Chinese developments in computer technology. Many photographs of Soviet machines, components, people, and places are included. First-hand information is also given on the BESM-I, BESM-II, Strela, Ural, and Kiev computers, plus several other machines. Machine specifications are presented in chart form, facilitating comparisons; op codes are given for the Ural-1 and Ural-2. 205 pp. Illus.

2. Feigenbaum, E. A., Soviet Cybernetics and Computer Sciences, 1960, RM-2799-PR, October 1961. Reprinted in IRE Transactions on Electronic Computers, Vol. EC-10, No. 4, December 1961.

   A description of the author's experiences as a delegate to the International Congress on Automatic Control, held in Moscow, June 27-July 7, 1960. The Memorandum discusses: (1) certain aspects of the conference; (2) some Soviet research projects in artificial intelligence and biocybernetics; and (3) general Soviet attitudes, techniques, and directions in the cybernetic and computer-related sciences. It is concluded that Soviet research in the computer sciences lags behind Western developments, but that

the gap is neither large nor based on a lack of understanding of fundamental principles. The Soviets will progress rapidly if and when priority, in terms of accessibility to computing machines, is given to their research. 77 pp. Illus.

3. Krieger, F. J., Soviet Philosophy, Science, and Cybernetics, RM-3619-PR, April 1963.

   A discussion of how all aspects of science--i.e., knowledge--are made to conform to the ideological mold of Marxism-Leninism in the Soviet Union. The larger part of the Memorandum consists of a thematic plan from the Soviet journal Questions of Philosophy [Voprosy filosofii], which lists over 300 topics suggested for discussion and study in the Soviet-planned society. 27 pp.

4. Ware, Willis H., and Wade B. Holland (eds.), Soviet Cybernetics Technology: I. Soviet Cybernetics, 1959-1962, RM-3675-PR, June 1963.

   Seven sets of translations in the area of Soviet cybernetics, together with commentary and analyses on the status of cybernetics in the Soviet Union and the direction of Soviet cybernetics research. This volume is concerned with general computer technology and cybernetics applications, rather than with specific machines. Particular emphasis was placed on selecting items for translation that survey the activities of organizations and conferences, and the current literature. 104 pp. Illus.

5. Ware, Willis H., and Wade B. Holland (eds.), Soviet Cybernetics Technology: II. General Characteristics of Several Soviet Computers, RM-3797-PR, August 1963.

   Several sets of translations detailing specifications for the Ural-2, Ural-4, BESM-II, Razdan-2, MN-10 and MN-14, Luch, and EPOS computers. The level of detail varies widely among the several articles, which were taken from such diverse sources as specification brochures, items in the popular press, technical journals, etc. Included is a set of instructions for the BESM-II which is quite dissimilar to that presented in Elements of Programming (see Vol. III in this series). 67 pp. Illus.

6. Ware, Willis H., and Wade B. Holland (eds.), Soviet Cybernetics Technology: III. Programming Elements of the BESM, Strela, Ural, M-3, and Kiev Computers, Translated by A. S. Kozak, RM-3804-PR, September 1963.

   A translation from the Russian book Elements of Programming, detailing the instruction formats for five of the better known Soviet digital computers. Some notes are included to help place the machines in perspective. Specially-prepared charts give the operation codes for the five machines, along with the original Russian terminology and its English translation. 91 pp. Illus.

7. Levien, Roger, and M. E. Maron, Cybernetics and Its Development in the Soviet Union, RM-4156-PR, July 1964.

   An introduction to the subject of cybernetics with special reference to its origins and ramifications in the United States and its subsequent development in the Soviet Union. Intended for non-experts in the field, it attempts to provide a sufficient non-technical background to facilitate appreciation of the potential impact of cybernetics on science and society. The survey of Soviet cybernetics reveals the intense interest and activity in the Soviet Union, pointing out how scientific research, military applications, economic planning, education, industry, etc., are affected by developments in cybernetics. 35 pp.

8. Holland, Wade B., (ed. & trans.), Soviet Cybernetics Technology: IV. Descriptions of the MN-11, MN-M and MN-7 Analog Computers and of Three Miscellaneous Electronic Devices, RM-4461-PR, February 1965.

   A collection of translations detailing technical specifications of the three indicated Soviet analog computers, and of the BPZ-1 fixed-delay unit, the I-5 CRT indicator, and the VPRR-2 electronic device for controlling tooling modes. The translations have been made from equipment specification brochures prepared for use by the Soviet technical and scientific community and for use at exhibits and trade fairs. 22 pp. Illus.

have been indexed. Russian-English and English-Russian glossaries of all ALGOL and ALGEC terms are appended. (The version of ALGEC translated in this Memorandum is superseded by that contained in Part VIII, RM-5136-PR.) 158 pp.

12. Wirth, Niklaus, Soviet Cybernetics Technology: IX. ALGEC--Summary and Critique, RM-5157-PR (in preparation).

A summary, evaluation, and critique of the preliminary and final versions of the ALGEC programming language.