CONTROL DATA

CONTROL DATA®
6000 COMPUTER SYSTEMS

ALGOL 60 REFERENCE MANUAL
VERSION 3

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

REVISION RECORD

REVISION	DESCRIPTION
A	Original printing.
(6-18-71)	
B	Addition of Object-Time Diagnostics. Pages affected are: 8-16 thru 8-19. Comment Sheet.
(11-22-71)	
C	Addition of new diagnostics and new Appendix D, "Interactive ALGOL under INTERCOM", minor
(6-28-72)	corrections included. Pages affected: iii, vii; 1-1, 1-2; 2-11; 3-42; 4-7; 6-1, 6-2; 8-14, 8-16 thru 8-20; 14-1;
	A-1 thru A-5; C-1; D-1 thru D-4; Index 1 thru 9; Comment Sheet.
D	Technical corrections and to note use under SCOPE 3.4 and KRONOS 2.1. Pages affected: iii, v, ix;
(10-05-73)	1-3; 2-17, 2-33; 3-1, 3-38, 3-39, 3-45; 4-3, 4-4; 5-1; 6-1; 7-1, 7-4; 8-14; 9-2, 9-6; 10-1; 13-1; 15-3; A-1,
	A-2, A-5, A-6; D-1, D-9; Comment sheet

Publication No.
60329000

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Software Documentation
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

© 1971, 1972, 1973
Control Data Corporation
Printed in the United States of America

PREFACE

This manual describes the ALGOL-60 language (Version 3) for the CONTROL DATA® CYBER 70/Models 72, 73, 74 or 6000 Series computers. It is assumed that the reader has knowledge of an existing ALGOL language and the CONTROL DATA 6000 Series computer systems.

ALGOL Version 3.1 operates under control of the SCOPE 3.3 or SCOPE 3.4 operating systems. Interactive compilation and execution is provided under INTERCOM for SCOPE users.

ALGOL Version 3.0 operates under KRONOS® 2.1.

Related manuals in which the ALGOL user may find additional information are:

	Publication No.
SCOPE 3.3 Users Guide	60252700
SCOPE 3.3 Reference Manual	60305200
SCOPE 3.4 Reference Manual	60307200
LOADER Reference Manual	60344200
INTERCOM 3 Reference Manual	60258000
INTERCOM 4 Reference Manual	60307100
KRONOS 2.1 Reference Manual	60407000

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or undefined parameters.

CONTENTS

	INTRODUCTION	ix
CHAPTER 1	ALGOL SYSTEM DESCRIPTION	1-1
	1.1 Compiler Features	1-1
	1.2 Compiler Package	1-2
	1.3 Compiler Structure	1-2
	1.4 Library Subprograms	1-3
	1.5 Operating System Interface	1-3
	1.6 Machine Configuration	1-3
CHAPTER 2	LANGUAGE COMPARISON WITH THE ALGOL-60 REVISED REPORT	2-1
	2.1 Language Conventions	2-1
	Revised Report on the Algorithmic Language ALGOL-60	2-2
	Introduction	2-4
	1. Structure of the Language	2-7
	1.1 Formalism for Syntactic Description	2-7
	2. Basic Symbols, Identifiers, Numbers, and Strings Basic Concepts	2-8
	2.1 Letters	2-8
	2.2 Digits	2-8
	2.3 Delimiters	2-9
	2.4 Identifiers	2-10
	2.5 Numbers	2-10
	2.6 Strings	2-12
	2.7 Quantities, Kinds and Scopes	2-13
	2.8 Values and Types	2-13
	3. Expressions	2-13
	3.1 Variables	2-14
	3.2 Function Designators	2-15
	3.3 Arithmetic Expressions	2-17
	3.4 Boolean Expressions	2-22
	3.5 Designational Expressions	2-24
	4. Statements	2-25
	4.1 Compound Statements and Blocks	2-25
	4.2 Assignment Statements	2-27
	4.3 Go To Statements	2-28
	4.4 Dummy Statements	2-29
	4.5 Conditional Statements	2-29
	4.6 For Statements	2-31
	4.7 Procedure Statements	2-33

5. Declarations 2-35
5.1 Type Declarations 2-36
5.2 Array Declarations 2-38
5.3 Switch Declarations 2-39
5.4 Procedure Declarations 2-40

Examples of Procedure Declarations 2-46

Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49

CHAPTER 3 INPUT-OUTPUT 3-1

3.1 Comparison with ACM Proposal for Input-Output 3-1
A Proposal for Input-Output Conventions in ALGOL-60 3-2

A. Formats 3-2
A.1 Numbered Formats 3-2
A.2 Other Formats 3-6
A.3 Format Strings 3-11
A.4 Summary of Format Codes 3-12
A.5 "Standard" Format 3-12
B. Input and Output Procedures 3-13
B.1 General Characteristics 3-13
B.2 Horizontal and Vertical Control 3-16
B.3 Layout Procedures 3-18
B.4 List Procedures 3-21
B.5 Input and Output Calls 3-22
B.6 Control Procedures 3-36
B.7 Other Procedures 3-37
C. Example 3-38
D. Machine-dependent Portions 3-39
3.2 Additional Input-Output Procedures 3-41
3.3 Control Procedures 3-41
3.4 Hardware Function Procedures 3-42
3.5 Miscellaneous Procedures 3-43
3.6 Input-Output Errors 3-44
3.7 End-of-File 3-44
3.8 End-of-Tape 3-44
3.9 ECS Procedures 3-45
3.10 Interactive Procedures 3-45

CHAPTER 4 INPUT TO COMPILATION 4-1

4.1 Source Program Definition 4-1
4.2 Source Procedure Definition 4-2
4.3 Source Input Restrictions 4-3
4.4 Language Conventions 4-3
4.5 Card Conventions 4-4
4.6 Source Deck 4-4

CHAPTER 5	OUTPUTS FROM COMPILATION	5-1
	5.1 Binary Output	5-1
	5.2 Assembly-Language Object Code	5-3
	5.3 Source Listing	5-4
CHAPTER 6	ALGOL CONTROL CARD	6-1
	6.1 Input-Output Options	6-1
	6.2 Compilation and Execution Speed	6-4
CHAPTER 7	CHANNEL CARDS	7-1
	7.1 Channel Define Card	7-1
	7.2 Channel Equate Card	7-3
	7.3 Channel End Card	7-3
	7.4 Duplication of Channel Numbers	7-3
	7.5 Duplication of File Names	7-4
	7.6 Standard ALGOL Channel Cards	7-4
	7.7 Typical Channel Cards	7-4
CHAPTER 8	ALGOL DIAGNOSTICS	8-1
	8.1 Compiler Diagnostics	8-1
	8.2 Compile-Time and Object-Time I/O Diagnostics	8-14
	8.3 Object-Time Diagnostics	8-15
CHAPTER 9	COMPILER DESCRIPTION	9-1
	9.1 Information Flow	9-1
	9.2 Language Translation	9-1
	9.3 Language Analysis	9-1
	9.4 Identifier (Symbol) Table	9-2
	9.5 Compiler Subprogram Descriptions	9-2
	9.6 Overall Compiler Flow	9-5
CHAPTER 10	OBJECT PROGRAM	10-1
	10.1 Run-Time Supervisory Program	10-1
	10.2 Object-Code Structure	10-1
	10.3 Object-Code Generation	10-1
	10.4 Library Subprograms	10-2
	10.5 Address-Field Conventions	10-2
CHAPTER 11	OBJECT-TIME STACK	11-1
	11.1 Stack Structure	11-1
	11.2 Stack Entries	11-4
	11.3 Details of Descriptions	11-6

CHAPTER 12	OBJECT-TIME ABNORMAL TERMINATION DUMP	12-1
	12.1 Structured Dump	12-1
	12.2 Global and Environmental Information	12-1
CHAPTER 13	EXECUTION TIME OPTIONS	13-1
	13.1 Stack Statistics (S)	13-2
	13.2 Debugging Output (D and C)	13-2
	13.3 Abnormal Termination Dump Formats (E)	13-2
	13.4 Examples	13-3
CHAPTER 14	DEBUGGING FACILITIES	14-1
	14.1 General	14-1
	14.2 Debugging Directives	14-1
	14.3 Trace Directive	14-2
	14.4 Snap Directive	14-3
	14.5 Off Directive	14-3
	14.6 Debugging Output	14-5
	14.7 Label Monitoring	14-5
	14.8 Procedure Monitoring	14-6
	14.9 Simple Variable Monitoring	14-6
	14.10 Array and Subscripted Variable Monitoring	14-8
CHAPTER 15	ALGOL MACROS	15-1
	15.1 Interface Macros	15-1
	15.2 Entry-Exit Macros	15-2
	15.3 Specification Macros	15-4
	15.4 Formal Handling Macros	15-6
APPENDIX A	STANDARD SCOPE CHARACTER SETS AND ALGOL CHARACTER SETS	A-1
APPENDIX B	SAMPLE PROGRAM	B-1
APPENDIX C	CHARACTER REPRESENTATION OF ALGOL SYMBOLS	C-1
APPENDIX D	INTERACTIVE ALGOL UNDER INTERCOM	D-1
APPENDIX E	M OPTION STORAGE ALLOCATION	E-1
INDEX		Index-1

INTRODUCTION

This reference manual presents the rules and details involved in writing a program in the ALGOL language; it includes sufficient information to prepare, compile, and execute such a program.

Control Data ALGOL closely conforms to the definition of the international algorithmic language ALGOL published in The Communications of the ACM, 1963, vol. 6 no. 1, pp. 1-17; "The Revised Report on the Algorithmic Language, ALGOL-60", and the input-output procedures provided as additions to the language in "A Proposal for Input-Output Conventions in ALGOL-60" published in The Communications of the ACM, vol. 7 no. 5, May 1964.

Control Data's input-output procedures incorporate many of the features recommended by the International Organization for Standardization in its Draft Proposal on the Algorithmic Language ALGOL, Appendix C "Proposal for Input-Output Procedures for ALGOL-60 (ACM)".

The ALGOL-60 Revised Report is presented in its entirety, and wherever Control Data's implementation of the language differs from the Report a full explanation of the differences are listed for all systems. The ALGOL-60 Revised Report is printed in bold type, and the explanation of the differences are in standard type. The name ALGOL means this implementation of ALGOL, unless otherwise specified.

The reader is referred to the following representative bibliography:

Baumann, R., Feliciano, M., Bauer, F. L., Samelson, K. Introduction to ALGOL, Prentice-Hall, Inc., 1964.

Dijkstra, E.W., A Primer of ALGOL-60 Programming, Academic Press, 1962.

Ekman, T., and Froberg, C.E.: Introduction to ALGOL Programming, Oxford University Press, 1965.

McCracken, Daniel D., A guide to ALGOL Programming, John Wiley & Sons, Inc., 1962.

ALGOL SYSTEM DESCRIPTION 1

1.1 COMPILER FEATURES

The ALGOL compiler for the 6000 Series computers is based in design on the ALGOL compiler developed by Regnecentralen, Copenhagen, Denmark, for the GIER computer. This design was adopted and, to some degree, extended by CONTROL DATA to provide the most generally advantageous features for an ALGOL compiler.

These features include:

Implementation of the complete ALGOL-60 revised language (wherever feasible and not in conflict with other advantages)

Comprehensive input-output procedures

Extensive compile-time and object-time diagnostics

Fast compilation

Wide variety of compilation options, such as the ability to compile both ALGOL programs and ALGOL procedures

Ability to generate and execute the object program in either segmented or non-segmented form

Interactive compilation option for use with INTERCOM (see appendix D)

Execution Breakpoint facility when executing under INTERCOM

MEMORY USAGE

The compiler attempts to compile every source text entirely within available memory with no reference to input-output devices. All intermediate information between the passes of the compiler is first stored in the compiler work areas. If the work areas are too small to contain all the intermediate information, the information is written onto scratch units and read back in by the next pass.

SOURCE INPUT

Source input is normally the card deck following the control card which calls for the ALGOL compiler. The source may also be specified from a different device by a control card option. Source input can consist of both ALGOL source programs and ALGOL source procedures. More than one source program and/or source procedure may be compiled with a single call.

COMPILE-TIME ERROR DETECTION

The compiler detects all source language infringements, and prints a diagnostic for each. The compiler also incorporates further checking into the object program to detect program errors which can be found only at execution time. All compilations proceed to the end of the source deck with normal error checking regardless of the occurrence of a source language error; but object code generation is suppressed if any errors are detected during compilation.

For the INTERCOM user this facility is combined with the normal editing facilites to provide line-by-line syntax checking. See Appendix D for more details.

COMPILER OUTPUTS

Compiler output is normally printed on the standard system output file. Output also may be requested on a different device with a control card option. The programmer may request the object code in segmented or non-segmented form.

OBJECT PROGRAM EXECUTION

In segmented form, a program can be loaded as part of the same compilation or later by the last pass of the compiler. In non-segmented form, a program is in standard relocatable binary format which can be loaded either in the same compilation or later by the system loader.

Execution of the object program, segmented or non-segmented, is controlled by a supervisory program external to the generated program.

The INTERCOM user can use the H option and get control at selected breakpoints to interactively monitor the execution of the program. See Appendix D.

OBJECT ERROR DETECTION

The object program includes code which detects errors not detected during compilation. Any error message is issued, a data map is printed, and the run is terminated. The data map displays current values of declared variables in a form easily related to the source program.

1.2 COMPILER PACKAGE

Many ALGOL subroutines are present in addition to those described below. For a complete list of all those included in the system library consult the installation handbook.

1.3 COMPILER STRUCTURE

ALGOL is the internal controller of the compiler and its main function is to load and pass control to each subprogram as required.

ALG0 processes the control card options delivered by the operating system.

ALG1 through ALG5 each form one pass of the compiler. Each subprogram overlays the previous one in a separate core-load. Each pass generates an intermediate form of the source text which is used as input to the next pass.

ALG1, ALG2, and ALG3 perform syntactic and semantic analysis of the source text.

ALG4 produces final output from the compiler, such as the object code in segmented or non-segmented form.

ALG5, although nominally part of the compiler, takes no part in the actual translation process; its only function is to control execution of the object program in segmented form.

1.4 LIBRARY SUBPROGRAMS

The library subprograms contain all standard procedures which can be called without prior declaration in an ALGOL source text. They also contain subprograms to perform object-time control functions external to the generated object program.

1.5 OPERATING SYSTEM INTERFACE

The compiler is designed to run under control of the SCOPE or KRONOS operating systems. Compilation is requested by a control card specifying the name ALGOL. This call results in the loading and execution of the subprogram ALGOL which controls the compilation process.

1.6 MACHINE CONFIGURATION

The basic machine configuration required for compilation consists of the minimum configuration required by SCOPE. In addition, when the source program generates more intermediate information than can be held in available memory, the compiler uses two scratch files to store this information.

The minimum number of words of available memory required by the compiler and its working areas is approximately 13K. With the minimum available memory, programs of a reasonable size can be compiled with no intermediate input-output. Additional available memory will permit compilation of larger programs entirely within memory.

LANGUAGE COMPARISON WITH THE ALGOL-60 REVISED REPORT

2

2.1 LANGUAGE CONVENTIONS

In this manual, ALGOL is described in terms of three languages: reference, hardware, and publication language, as indicated in the introduction to the ALGOL-60 Revised Report.

The reference language is computer independent and uses the basic ALGOL symbols, such as begin and end, to define the language syntax and semantics.

The hardware language is the representation of ALGOL symbols in characters acceptable to the computer; this is the language used by the programmer. For example, when the reference language calls for the basic ALGOL symbol begin, the programmer writes the seven hardware characters 'BEGIN'. The hardware representations of ALGOL symbols are shown in Table 1, Chapter 4.

Unless otherwise stated or implied, the basic ALGOL symbols (reference language) rather than their character equivalents (hardware language) are used consistently throughout this manual. This convention simplifies the explicit and implicit references to the ALGOL language as defined in the ALGOL-60 Revised Report.

For publication purposes only, the underlining convention delineates the basic ALGOL symbols. These symbols have no relation to the individual letters of which they are composed. Other than this convention, the publication language is not considered in this manual.

All descriptions of language modifications are made at the main reference in the Report; when feasible, language modifications are also noted at other points of reference. The reader should assume that modifications apply to all references to the features, noted or otherwise. If no comments appear at the main reference in the Report regarding language modifications to a particular section or feature, it is implemented in full accordance with the Report.

In addition to the language descriptions in this chapter, reserved identifiers which reference input-output procedure are described in Chapter 3.

The ALGOL-60 Revised Report as published in The Communications of the ACM, vol. 6, no. 1, pp 1-17 follows. Wherever CONTROL DATA's implementation of the language differs from the Report, the Report is printed first in boldface and the CONTROL DATA modification follows in standard type.

REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL–60†

Peter Naur (Editor)

J. W. Backus	C. Katz	H. Rutishauser	J. H. Wegstein
F. L. Bauer	J. McCarthy	K. Samelson	A. van Wijngaarden
J. Green	A. J. Perlis	B. Vauquois	M. Woodger

Dedicated to the memory of William Turanski.

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL-60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntatic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntatic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

†This report is published in The Communications of the ACM, in Numerische Mathematik, and in the Computer Journal.

CONTENTS

Introduction

1. Structure of the Language
   - 1.1 Formalism for syntatic description
2. Basic Symbols, Identifiers, Numbers, and Strings
   Basic Concepts
   - 2.1 Letters
   - 2.2 Digits. Logical values
   - 2.3 Delimiters
   - 2.4 Identifiers
   - 2.5 Numbers
   - 2.6 Strings
   - 2.7 Quantities, kinds and scopes
   - 2.8 Values and types
3. Expressions
   - 3.1 Variables
   - 3.2 Function designators
   - 3.3 Arithmetic expressions
   - 3.4 Boolean expressions
   - 3.5 Designational expressions
4. Statements
   - 4.1 Compound statements and blocks
   - 4.2 Assignment statements
   - 4.3 Go to statements
   - 4.4 Dummy statements
   - 4.5 Conditional statements
   - 4.6 For statements
   - 4.7 Procedure statements
5. Declarations
   - 5.1 Type declarations
   - 5.2 Array declarations
   - 5.3 Switch declarations
   - 5.4 Procedure declarations

Examples of Procedure Declarations

Alphabetic Index of Definitions of Concepts and Syntatic Units

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL†, as prepared at a conference in Zurich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL Bulletin, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represented the following organizations: Association Francaise de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Gennotschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the Communications of the ACM, where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM Communications. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives, from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960. Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference (Edited by M. Woodger)

A meeting of some of the authors of ALGOL-60 was held on April 2-3 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre.

† Preliminary report — International Algebraic Language. Comm ACM1, 12 (1958), 8.
Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages, edited by A.J. Perlis and K. Samelson. Num, Math. 1 (1959), 41-60.

The following were present:

Authors	Advisers	Observer
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman IFIP TC 2.1 Working Group ALGOL)
J. Green	R. Franciotti	
C. Katz	P. Z. Ingerman	
R. Kogon (representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL-60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in ALGOL Bulletin No. 14 were used as a guide.

(This report constitutes a supplement to the ALGOL-60 Report which should resolve a number of difficulties therein). Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions

2. The call by name concept

3. own: static or dynamic

4. For statement: static or dynamic

5. Conflict between specification and declaration

The authors of the ALGOL Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.

2. It is the defining language.

3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.

4. It is the basic reference and guide for compiler builders.

5. It is the guide for all hardware representations.

6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.

7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).

2. It is used for stating and communicating processes.

3. The characters to be used may be different in different countries but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.

2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.

3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language, and a language suitable for publications, among others, the following rules are recommended.

Reference Language	Publication Language
Subscript bracket [ ]	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	Raising of the exponent
Parentheses ( )	Any form of parentheses, brackets, braces
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

DESCRIPTION OF THE REFERENCE LANGUAGE

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets begin and end to form a compound statement.

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, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between begin and end constitutes 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 which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given.†

1.1 Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae.‡ Their interpretation is best explained by an example

$$< \text{ab} > ::= (\mid [\mid < \text{ab} > (\mid < \text{ab} > < \text{d} >$$

Sequences of characters enclosed in the brackets $< >$ represent metalinguistic variables whose values are sequences of symbols. The mark ::= and | (the latter with the meaning of or) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a a recursive rule for the formation of values of the variable $< \text{ab} >$. It indicates that $< \text{ab} >$ may have the value (or [or

†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, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

‡Cf. J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris, June 1959.

that given some legitimate value of < ab >, another may be formed by following it with the character (or by following it with some value of the variable < d >. If the values of < d > are the decimal digits, some values of < ab > are:

```
[(((1(37(
(12345(
( ( (
[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 < > as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

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 > ::= < letter > | < digit > | < logical value > | < 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

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

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value or delimiter). Letters do not have individual meaning. They are used for forming identifiers and strings† (Cf. sections 2.4 Identifiers, 2.6 Strings).

2.1 Letters

Since there is hardware representation for upper case letters only, lower case letters have no meaning.

2.2.1 Digits

< digit > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

†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). These are understood to have no relation to the individual letters of which they are composed. Within the present report (not including headings) underlining will be used for no other purpose.

2.2.2 Logical Values

< logical value > ::= true | false

The logical values have a fixed obvious meaning.

2.3 Delimiters

< delimiter > ::= < operator > | < separator > | < bracket > | < declarator > | < specificator >

< operator > ::= < arithmetic operator > | < relational operator > | < logical operator > |

< sequential operator >

< arithmetic operator > ::= + | − | × | / | ÷ | ↑

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

< logical operator > ::= ≡ | ⊃ | ∨ | ∧ | ¬

< sequential operator > ::= go to | if | then | else | for | do[†]

< separator > ::= , | . | 10 | : | ; | :=| ␣ | step | until | while | comment

< bracket > ::= (|) | [|] | ' | ' | begin | end

< declarator > ::= own | Boolean | integer | real | array | switch | procedure

< specificator > ::= string | label | value

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

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading. For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent to:
; comment < any sequence not containing; > ;	;
begin comment < any sequence not containing; > ;	begin
end < any sequence not containing end or ; or else >	end

By equivalence is here meant that any of the three structures shown in the left hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

[†]do is used in for statements. It has no relation whatsoever to the do of the preliminary report, which is not included in ALGOL-60.

2.3 Delimiters

The symbol code, defined below, is added to the language to permit reference to separately compiled procedures (Section 5.4.6).

<code procedure body indicator>: : = code

2.4 Identifiers

2.4.1 Syntax

< identifier > ::= < letter > | < identifier > < letter > | < identifier > < digit >

2.4.2 Examples

q
Soup
V 17a
a34kTM Ns
MARILYN

2.4.3 Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4 Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf., section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.4.3 Semantics

The maximum size of an identifier is 256 hardware characters. If a longer identifier is specified, only the first 256 characters are used.

The maximum number of identifiers which can be handled by the compiler without causing identifier table overflow depends on the identifier sizes and the amount of memory available to the compiler.

The compiler itself can handle up to 3583 unique identifiers. Identifier table overflow generally occurs before this number is reached.

2.5 Numbers

2.5.1 Syntax

< unsigned integer > ::= < digit > | < unsigned integer > < digit >

< integer > ::= < unsigned integer > | + < unsigned integer > |

- < unsigned integer >

< decimal fraction > :: = . < unsigned integer >

< exponent part > :: = 10 < integer >

< decimal number > :: = < unsigned integer >| < decimal fraction >|

< unsigned integer > < decimal fraction >

< unsigned number > :: = < decimal number >| < exponent part >|

< decimal number > < exponent part >

< 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 part is a scale factor expressed as an integral power of 10.

2.5.3 Semantics

A number has the format

$$d_1 d_2 \ldots d_j . d_{j+1} d_{j+2} \ldots d_n 10^{\pm e_1, e_2 \ldots e_m}$$

where the decimal point and the exponent field may or may not be explicit. If the decimal point is not explicit, it is assumed to follow the digit d_n (j=n). If the exponent field is not explicit, zero value is assumed. If the sign of the exponent field is not explicit, a positive exponent is assumed. Thus, all numbers are considered to have the same format and are treated identically.

If the magnitude of the exponent field exceeds 9999, the diagnostic NUMBER SIZE is issued.

A number is modified in three steps before it is converted to its final internal representation (Section 5.1.3).

1. All leading zeros are eliminated, including any following the decimal point.

2. Beginning with the first non-zero digit, the digits following the fourteenth are discarded. The number of digits discarded is added to the value of the exponent field.

3. The effect of the decimal point is incorporated by subtracting n-j (number of digits to the right of the point) from the value of the exponent field.

These three modifications effectively produce a number of the form $d_i d_{i+1} \ldots d_k 10^e$ where d_i is the first non-zero digit in the original number. If no non-zero digit is found, the number is given the internal value 0. d_{i+1}, d_{i+2}, etc., are the digits (zero or non-zero) immediately following d_i in the original number.

The last significant digit, d_k, is d_n if $n < i+13$ or is d_{i+13}.

The resultant exponent field value, e, is given by:

$$e = e_1 e_2 \ldots e_m - (n-j) + \max(0, n - (i+13)).$$

If e is greater than the maximum allowed decimal exponent, the diagnostic NUMBER SIZE is issued. If e is smaller than the minimum allowed decimal exponent, the number is given the internal value 0.

2.5.4 Types

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

2.5.4 Types

During compilation, numbers are flagged as type real or integer, according to the following rules:

Any number with an explicit decimal point and/or an explicit exponent part is flagged real.

All other numbers are flagged integer.

real and integer numbers are represented internally in the same form as real and integer variables (Section 5.1.3).

During object code generation, the type and value of a number may be changed depending on the operation and type of operand with which it is associated in the source program (Sections 3.3.4 and 4.3.4).

2.6 Strings

2.6.1 Syntax

< proper string >::=< any sequence of basic symbols not containing ' or '>|< empty >

< open string >::=< proper string >|'<open string >'|

< open string >< open string >

< string >::= '< open string >'

2.6.2 Examples

'5k, ,-'[[['∧=/:'Tt"
'. . This ␣ is ␣ a ␣'string"

2.6.1 Syntax

A proper string is defined as follows:

<proper string>::=<empty> | <any sequence of basic 6-bit display BCD characters not containing the symbols ' or ' or the character associated with external BCD 00_8>

2.6.3 Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ' and ' are introduced. The symbol ␣ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.6.3 Semantics

The string quote symbols 'and ' are introduced to enable the language to handle arbitrary sequences of basic characters (not basic symbols, as defined in the Report). These symbols are represented by the three-character sequences '(' and ')'. (Table 1, Chapter 4).

2.7 Quantities, Kinds and Scopes

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

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8 Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf, section 3.1.4.1).

The various "types" (integer, real, Boolean) basically denote properties of values. 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, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

< expression > ::= < arithmetic expression >| < Boolean expression >| < designational expression >

3.1 Variables

3.1.1 Syntax

< variable identifier > :: = < identifier >

< simple variable > :: = < variable identifier >

< subscript expression > :: = < arithmetic expression >

< subscript list > :: = < subscript expression > | < subscript list > , < subscript expression >

< array identifier > :: = < identifier >

< subscripted variable > :: = < array identifier > [< subscript list >]

< variable > :: = < simple variable > | < subscripted variable >

3.1.2 Examples

```
epsilon
detA
a17
Q [7,2]
x [sin(nXpi/2), Q [3, n, 4]]
```

3.1.3 Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2 Array Declarations).

3.1.4 Subscripts

3.1.4.1 Subscripted variables designate values which are components of multidimensional arrays (cf. 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 value of its subscripts (cf. 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 is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2 Array Declarations).

3.1.4.2

No check is made to ensure that the value formed by assigning each subscript expression to a fictitious integer variable (Section 4.2.4) is within the corresponding bounds of the array. However, the address of the referenced array component (combination of all of the fictitious integer variables) is checked to ensure that it lies within the bounds of the complete array. This array bounds check is performed only with the option F (Section 6.1, Chapter). The final address of the referenced array component is assumed to be a normal machine word address.

3.1.4.3

The Q option (Chapter 6) results in optimization of the code compiled for for loops and array addressing. Components of a subscript expression containing variables and constants will be compiled at the head of the current for loop, rather than within it unless the expression contains a variable which occurs in the current for loop as:

a left part

or an actual parameter in a call (procedure statement or function designator).

If a call within a for loop changes the value of a variable accessible to both procedure and for loop and that variable is not an actual parameter of the procedure, then subscript expressions in the for loop which depend on the variable will be evaluated incorrectly unless the variable is subject to a side effect elsewhere in the for loop as described above.

This restriction applies only to the Q option.

3.2 Function Designators

3.2.1 Syntax

< procedure identifier > :: = < identifier >

< actual parameter > :: = < string > | < expression > | < array identifier > |

< switch identifier > | < procedure identifier >

< letter string > :: = < letter > | < letter string > < letter >

< parameter delimiter > :: = , |) < letter string > : (

< actual parameter list > :: = < actual parameter > |

< actual parameter list > < parameter delimiter > < actual parameter >

< actual parameter part > :: = < empty > | (< actual parameter list >)

< function designator > :: = < procedure identifier > < actual parameter part >

3.2.2 Examples

```
sin (a-b)          S(s-5) Temperature: (T) Pressure: (P)
J (v+s,n)          Compile(' := ')Stack:(Q)
R
```

3.2.3 Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the values of a function designator.

3.2.4 Standard Functions

Certain identifiers should be reserved for the standard functions of analysis, which will be 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)

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

arctan(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) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.4 Standard Functions

All input-output functions (Chapter 3) are expressed as calls of standard procedures. The list of reserved identifiers is expanded to include the names of these and other utility procedures:

INLIST	PUTTARAY	STRINGELEMENT	BACKSPACE
OUTLIST	CLOCK	CHLENGTH	IOLTH
INPUT	HLIM	ARTHOFLW	MODE
OUTPUT	VLIM	BADDATA	CONNECT
INREAL	HEND	PARITY	DUMP
OUTREAL	VEND	EOF	READECS
INARRAY	NODATA	REWIND	WRITEECS
OUTARRAY	TABULATION	UNLOAD	CONNEC
INCHARACTER	FORMAT	SKIPF	DISCON
OUTCHARACTER	SYSPARAM	SKIPB	
GETARRAY	EQUIV	ENDFILE	

Calls to all standard procedures (input-output and function) conform to the syntax of calls to declared procedures (Section 4.7.1) and in all respects are equivalent to regular procedure calls. This specifically includes the use of a standard procedure identifier as an actual parameter in a procedure call.

If a standard procedure is not needed throughout a program, its identifier may be declared to have another meaning at any level; the identifier assumes the new meaning rather than that of a standard procedure.

Since all standard procedures are contained on the system library, they are available to any object program (Chapter 5).

3.2.5 Transfer Functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely,

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3 Arithmetic Expressions

3.3.1 Syntax

< adding operator > ::= + | −

< multiplying operator > ::= × | / | ÷

< primary > ::= < unsigned number > | < variable > |

< function designator > | (< arithmetic expression >)

< factor > ::= < primary > | < factor > ↑ < primary >

< 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 > < simple arithmetic expression > else

< arithmetic expression >

3.3.2 Examples

Primaries:

```
7.394₁₀–8
sum
w [i+2,8]
cos (y+z×3)
(a–3/y+vu↑8)
```

Factors:

```
omega
sum ↑cos(y+z×3)
7.394₁₀–8 ↑w[i+2,8] ↑(a–3/y+vu↑8)
```

Terms:

```
U
omega×sum↑cos(y+z×3)/7.349₁₀–8↑w [i+2,8] ↑
  (a–3/y+vu↑8)
```

Simple arithmetic expression:

```
U–Yu+omega×sum↑cos(y+z×3)/7.349₁₀–8↑w[i+2,8] ↑
  (a–3/y+vu↑8)
```

Arithmetic expressions:

```
w×u–Q(S+Cu)↑2
if q>0 then S+3×Q/A else 2×S+3×q
if a<0 then U+V else if a×b>17 then U/V else if k≠y then V/U else 0
a×sin(omega×t)
0.57₁₀12×a[N×(N–1)/2,0]
(A×arctan(y)+Z)↑(7+Q)
if q then n–1 else n
if a<0 then A/B else if b = 0 then B/A else z
```

3.3.3 Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary 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 (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value true is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood).

The construction:

else < simple arithmetic expression >

is equivalent to the construction:

else if true then < simple arithmetic expression >

3.3.3 Semantics

If during the evaluation of an arithmetic expression, a machine arithmetic error condition (overflow, underflow or division fault) arises, caused for example by an attempt at division by 0, an error condition exists in the object program. If the procedure ARTHOFLW (Chapter 3) has not been called, or if a label established by it is no longer accessible, the object program terminates abnormally with the message ARITHMETIC OVERFLOW. If an arithmetic overflow label has been established, control passes to it.

3.3.4 Operators and types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead 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 expression will be integer if both of the operands are of integer type, otherwise real.

3.3.4.2 The operations < term >/< factor > and < term > ÷ < factor > both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

a/b×7/(p−q) ×v/s

means

$$((((a \times (b^{-1})) \times 7) \times ((p-q)^{-1})) \times v) \times (s^{-1})$$

The operator / is defined for all four combinations of types real and integer and will yield results of real type in any case. The operator ÷ is defined only for two operands both of type integer and will yield a result of type integer, mathematically defined as follows:

$$a \div b = sign(a/b) \times entier(abs(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4 Operators and Types

When the type of an arithmetic expression cannot be determined at compile-time, it is considered real. For example, the parenthesized expression in the following statement is considered real if one or both of the arithmetic expressions R and S is real:

P × (if Q then R else S)

If both operands in a simple arithmetic expression are numbers, the transformation (from type real to integer, or integer to real) and the operation itself are performed at compile-time. The type of the one resulting number is defined according to the number types and the particular operation involved.

If the result of an expression is assigned to a variable with a different type, the compiler generates the code to transform the result to the proper type (Section 4.2.4).

When the final result is a number, transformation is performed at compile-time (as in the assignment of a simple number to a variable of a different type).

The internal representations of type real and integer values and the transformations between them are described in Section 5.1.3.

3.3.4.3 The operation < factor > ↑ < primary > denotes exponentiation, where the factor is the base and the primary 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 integer type, r for a number of real type, and a for a number of either integer or real type, the result is given by the following rules:

a↑i If i>0, a×a×...×a(i times), of the same type as a.

If i = 0, if a≠0,1, of the same type as a.

if a = 0, undefined.

If i<0, if a≠0,1/(a×a×...×a) (the denominator has −i factors), of type real.

if a = 0, undefined.

a↑r If a>0, exp(r× ln(a)), of type real.

If a = 0, if r>0,0.0, of type real.

if r≤0, undefined.

If a<0, always undefined.

3.3.4.3

The rule for evaluating an expression of the form a↑i or a↑r is the same as defined above except when a is of type integer and i is an integer variable with a positive value. In this case, the result is real, whereas the Report defines it as integer. (If i is a positive integer number, however, the result is integer as defined.)

3.3.5 Precedence of operators

The sequence of operations within one expression is generally 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: ↑
second: ×/÷
third: +−

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 Arithmetics 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. No exact arithmetic will be specified, however, 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 methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4 Boolean Expressions

3.4.1 Syntax

< relational operator > :: = < | ⩽ | = | ⩾ | > | ≠

< relation > :: = < simple arithmetic expression >

< relational operator > < simple arithmetic expression >

< Boolean primary > :: = < logical value > | < variable > |

< function designator > | < relation > | (< Boolean expression >)

< Boolean secondary > :: = < Boolean primary > | ¬ < Boolean primary >

< Boolean factor > :: = < Boolean secondary > |

< Boolean factor > ∧ < Boolean secondary >

< Boolean term > :: = < Boolean factor > | < Boolean term >

∨ < Boolean factor >

< implication > :: = < Boolean term > | < implication > ⊃ < Boolean term >

< simple Boolean > :: = < implication > |

< simple Boolean > ≡ < implication >

< Boolean expression > :: = < simple Boolean > |

< if clause > < simple Boolean > else < Boolean expression >

3.4.2 Examples

x = –2
Y > V ∨ z < q
a+b > –5 ∧ z–d > q↑2
p ∧ q ∨ x≠y
g ≡ ¬a ∧ b ∧ ¬c ∨ d ∨ e ⊃ ¬f
if k < 1 then s > w else h ⩽ c
if if if a then b else c then d else f then g else h < k

3.4.3 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.

3.4.4 Types

Variables and function designators entered as Boolean primaries must be declared Boolean (cf. section 5.1.Type Declarations and section 5.4.4. Values of Function Designators).

3.4.5 The operators

Relations take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The meaning of the logical operators ¬(not), ∧ (and), ∨ (or), ⊃ (implies), and ≡ (equivalent), is given by the following function table.

b1	false	false	true	true
b2	false	true	false	true
¬ b1	true	true	false	false
b1 ∧ b2	false	false	false	true
b1 ∨ b2	false	true	true	true
b1 ⊃ b2	true	true	false	true
b1 ≡ b2	true	false	false	true

3.4.6 Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1 According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5

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

third: ¬

fourth: ∧

fifth: ∨

sixth: ⊃

seventh: ≡

3.4.6.2 The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5 Designational Expressions

3.5.1 Syntax

< label > :: = < identifier >| < unsigned integer >

< switch identifier > :: = < identifier >

< switch designator > :: = < switch identifier > [< subscript expression >]

< simple designational expression > :: = < label >| < switch designator >|

(< designational expression >)

< designational expression > :: = < simple designational expression >|

< if clause > < simple designational expression > else

< designational expression >

3.5.2 Examples

17
p9
Choose [n-1]
Town [if y < 0 then N else N+1]
if Ab < c then 17 else q [if w ≤ 0 then 2 else n]

3.5.3 Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3 Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4 The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. 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.

3.5.5 Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

3.5.5 Unsigned Integers as Labels

Integer labels are not permitted.

4. Statements

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

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

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

4.1 Compound Statements and Blocks

4.1.1 Syntax

< unlabelled basic statement > :: = < assignment statement >|

< go to statement >| < dummy statement >| < procedure statement >

< basic statement > :: = < unlabelled basic statement >| < label > : < basic statement >

< unconditional statement > :: = < basic statement >|

< compound statement >| < block >

< statement > :: = < unconditional statement >|

< conditional statement >| < for statement >

< compound tail > :: = < statement > end| < statement > ;

< compound tail >

< block head > :: = begin < declaration >| < block head > ;

< declaration >

< unlabelled compound > :: = begin < compound tail >

< unlabelled block > :: = < block head > ; < compound tail >

< compound statement > :: = < unlabelled compound >|

< label > : < compound statement >

< block > :: = < unlabelled block >| < label > : < block >

< program > :: = < block >| < compound statement >

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

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

Block:

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

S end

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

4.1.2 Examples

Basic Statements:

```
a :=p+q
go to Naples
START: CONTINUE:W:=7.993
```

Compound Statement:

```
begin x :=0 ;for y:=1 step 1 until n do
        x :=x+A [y] ;
        if x > q then go to STOP else if x > w-2 then go to S;
        Aw:St:W :=x+bob end
```

Block:

```
Q: begin integer i,k ;real w ;
for i : = 1 step 1 until m do
for k: = i+1 step 1 until m do
begin w : = A [i,k] ;
    A [i,k] :=A[k,i] ;
    A [k,i] :=w end for i and k
end block Q
```

4.1.1 Syntax

Replace the definition of <program> with:

<program>::=<unlabeled block> | <unlabeled compound>

4.1.3 Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. 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 block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e., will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e., labelling 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 of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.1.3 Semantics

Blocks may be nested to a maximum of 32 levels.

4.2 Assignment Statements

4.2.1 Syntax

< left part > :: = < variable > : = | < procedure identifier > : =

< left part list > :: = < left part > | < left part list > < left part >

< assignment statement > :: = < left part list > < arithmetic expression > |

< left part list > < Boolean expression >

4.2.2 Examples

```
s : = p [0] : = n : = n+1+s
n : = n+1
A : = B/C-v-qXS
S [v,k+2] : = 3-arctan(sXzeta)
V : = Q > Y ∧ Z
```

4.2.3 Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1 Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3 The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4 Types

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is Boolean, the expression must likewise be Boolean. If the type is real or integer, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from real to integer type, the transfer function is understood to yield a result equivalent to

```
entier(E+0.5)
```

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.2.4 Types

If the type of an arithmetic expression (Section 3.3.4) is different from that of the variable or procedure identifier to which it is assigned, the compiler generates the code to perform the transformation from one type to the other.

If an expression results only in a number (Section 2.5.4), the transformation is done at compile-time, and the resulting number is flagged according to its new type.

The internal representations of real and integer values and the transformations between them are described in Section 5.1.3.

4.3 Go To Statements

4.3.1 Syntax

< go to statement > :: = go to < designational expression >

4.3.2 Examples

```
go to 8
go to exit [n+1]
go to Town [if y < 0 then N else N+1]
go to if Ab < c then 17 else q [if w < 0 then 2 else n]
```

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 to be 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 may, however, lead from outside into a compound statement.

4.3.5 Go to an undefined switch designator

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

4.3.5 Go to Undefined Switch Designator

When a go to statement is executed for a designational expression which is a switch with an undefined value, the object program terminates abnormally with the message SWITCH BOUNDS ERROR.

4.4 Dummy Statements

4.4.1 Syntax

< dummy statement > :: = < empty >

4.4.2 Examples

```
L:
begin . . . ; John: 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 v > u then V: q:=n+m else go to R
if s < 0 ∨ P ≤ Q then AA: begin if q < v then a := v/s
    else y := 2Xa end
    else if v > s then a :=v-q else if v > s-1
      then go to S
```

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. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

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

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

and

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

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

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value true is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e., the statement following the complete conditional statement. Thus the effect of the delimiter else may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

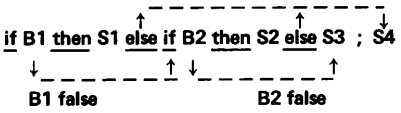
else < unconditional statement >

is equivalent to

else if true then < unconditional statement >

If none of the Boolean expressions of the if clause, is true, the effect of the whole conditional statement will be 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 else.

4.6 For statements

4.6.1 Syntax

< for list element > :: = < arithmetic expression >|

< arithmetic expression > step < arithmetic expression > until

< arithmetic expression >| < arithmetic expression > while

< Boolean expression >

< for list > :: = < for list element >| < for list > , < for list element >

< for clause > :: = for < variable > : = < for list > do

< for statement > :: = < for clause > < statement >|

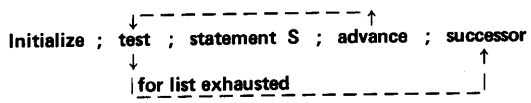
< label > : < for statement >

4.6.2 Examples

```
for q : = 1 step s until n do A [q] : = B [q]
for k : = 1, V1X2 while V1<N do
    for j : = I+G,L,1 step 1 until N,C+D do
        A [k,j] : = B [k,j]
```

4.6.3 Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition, it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment for the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.3 Semantics

If, in a for statement, the controlled variable is subscripted, the same array element is used as the control variable throughout the execution of the for statement, regardless of any changes that might occur to the value of the subscript expressions during its execution. The element used is the one referenced by the value of the subscript expressions on entry to 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 controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S 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 Step-until-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 may be described most concisely in terms of additional ALGOL statements as follows:

```
   V := A ;
L1:if (V-C)×sign (B) > 0 then go to element exhausted;
   statements S ;
   V := V+B ;
   go to L1 ;
```

where V is the controlled variable of the for clause and element exhausted points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3 While-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 ALGOL statements as follows:

```
L3: V := E ;
    if ¬ F then go to element exhausted ;
        Statement S ;
        go to L3 ;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5 The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

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

4.6.6 Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.6.6 Go to Leading into a For Statement

A go to statement, executed from outside a for statement not currently being executed, which refers to a label within the for statement, causes the object program to terminate abnormally with the message UNDEFINED FOR LABEL. This will not happen if an exit is made from the for statement prior to its end. However, no checking will occur in the object program if the source program was compiled with the Q option. (See paragraph 6.1)

4.7 Procedure Statements

4.7.1 Syntax

< actual parameter > ::= < string >|< expression >|< array identifier >|

< switch identifier >|< procedure identifier >

< letter string > ::= < letter >|< letter string > < letter >

< parameter delimiter > ::= ,|) < letter string > :(

< actual parameter list > ::= < actual parameter >|< actual parameter list >

< parameter delimiter > < actual parameter >

< actual parameter part > ::= < empty >|

(< actual parameter list >)

< 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,Yy,I,K)
Innerproduct (A [t,P,u] ,B [P],10,P,Y)
```

These examples correspond to examples given in section 5.4.2.

4.7.3 Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4 Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing 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 quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being 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 (cf. section 5.4.5). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.2 Name replacement (call by name)

Any formal parameter not quoted 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 will be 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 nonlocal 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 will be avoided through suitable systematic changes of the latter identifiers.

4.7.4 Actual-formal correspondence

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

4.7.5.1 If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL–60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2 A formal parameter which occurs as a left part variable is an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3 A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4 A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. Section 5.4.1) and which defines the value of a function designator (cf. Section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.5 Restrictions

A label cannot be specified by value.

There are no restrictions on the real or integer correspondence between formal and actual parameters. However, for formal parameters called-by-name, real to integer conversions are checked and performed only if the C option appears on the ALGOL control card. (Section 6.1, Chapter 6).

A maximum of 63 formal parameters may be included in a procedure declaration (Section 5.4.3); therefore, a maximum of 63 actual parameters may be included in a procedure call. No more than 62 constants may be used as actual parameters.

4.7.6 Deleted

4.7.7 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 the elaborate ones is entirely optional.

4.7.8 Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

4.7.8 Procedure Body Expressed in Code

The symbol code is included to permit reference to procedures which are compiled separately from the program or procedure in which they are referenced (Section 5.4.6).

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 for one block. Outside this block the particular identifier may be used for other purposes (cf. 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 had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

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

A declaration may be marked with the additional declarator own. This has the following effect: upon a re-entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax

< declaration > ::= < type declaration > | < array declaration > | < switch declaration > | < procedure declaration >

5.1 Type Declarations

5.1.1 Syntax

< type list > ::= < simple variable > |

< simple variable > , < type list >

< type > ::= real | integer | Boolean

< local or own type > ::= < type > | own < type >

< type declaration > ::= < local or own type > < type list >

5.1.2 Examples

```
integer p,q,s
own Boolean Acryl,n
```

5.1.3 Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values true and false.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of own, see the fourth paragraph of section 5 above.

5.1.3 Semantics

Variables of type real and integer are represented internally in a 60-bit floating-point form, with a 48-bit coefficient, sign bit, and 11-bit biased exponent, so that the range of non-zero real and integer variables is:

$$3.1 * 10 \uparrow (-294) \approx (2 \uparrow 48 - 1) * 2 \uparrow (-1022)$$

$$\left\langle \begin{matrix} \text{abs(real)} \\ \text{abs(integer)} \end{matrix} \right\langle$$

$$(2 \uparrow 48 - 1) * 2 \uparrow 1022 \approx 1.3 * 10 \uparrow 322$$

Real and integer values with up to 14 (and some with 15) significant decimal digits can be represented.

A zero real or integer value is represented by 60 zero bits in fixed-point form.

Real and integer numbers (Section 2.5.4) have the same range of values and are represented in the same form as real and integer variables. In the evaluation of arithmetic expressions (Section 3.3.4) and their assignment to variables of different type (Section 4.2.4), conversion from real to integer is performed by closed subroutines at compile-time and in line code at object-time. No conversion is required from integer to real because of their identical internal representations.

This conversion selects from the real value an integer value according to the rule:

ENTIER (real value + 0.5)

Variables of type Boolean are represented in 60-bit fixed-point form; only the high order bit is significant:

true ::= high-order bit = 1

false ::= high-order bit = 0

5.3.3 Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1,2, . . . , obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4 Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5 Influence of scopes

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4 Procedure Declarations

5.4.1 Syntax

< formal parameter > ::= < identifier >

< formal parameter list > ::= < formal parameter >|

< formal parameter list > < parameter delimiter >

< formal parameter >

< formal parameter part > ::= < empty >| (< formal parameter list >)

< identifier list > ::= < identifier >| < identifier list > , < identifier >

< value part > ::= value < identifier list > ; | < empty >

< specifier > ::= string | < type >| array| < type > array| label| switch|

procedure | < type > procedure

< specification part > ::= < empty >| < specifier > < identifier list > ; |

< specification part > < specifier > < identifier list > ;

< procedure heading > ::= < procedure identifier >

< formal parameter part > ; < value part > < specification part >

< procedure body > ::= < statement >| < code >

< procedure declaration >::=

procedure < procedure heading >< procedure body >|

< type > procedure < procedure heading > < procedure body >

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

```
procedure Spur (a) Order: (n) Result: (s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s+a [k,k]
end

procedure Transpose (a) Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i,k ;
for i := 1 step 1 until n do
    for k := 1+i step 1 until 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) size: (n,m) Result: (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 ; integer n,m,i,k ; real y ;
begin integer p,q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until 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:(y) ; value k ;
integer k,p ;real y,a,b ;
begin real s ;
s := 0 ;
for p := 1 step 1 until k do s := s+a×b ;
y := s
end Innerproduct
```

5.4.1 Syntax

The following definition of <code> is added:

<d> : : = <digit>

<code number> : : = <d> | <d><d> | <d><d><d> | <d><d><d><d> | <d><d><d><d><d>

<code> : : = code <code number>

5.4.3 Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which 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 will, whenever the procedure is activated (cf. section 3.2 Function Designators and section 4.7. Procedure Statements) be assigned the values of or be replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal 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 labelling 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.

5.4.3 Semantics

The maximum number of formal parameters permitted in the declaration of a procedure is 63. A source procedure (Chapter 4) may employ the same features as a procedure declared in a source program, except it may not be formally recursive. That is, there may be no occurrence of the procedure identifier within the body of the procedure other than as a left part in an assignment statement.

5.4.4 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, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5 Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.5 Specifications

The last sentence should be changed to read: "...and specifications of all formal parameters, if any, must be supplied."

5.4.6 Code as procedure body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

5.4.6 Code as a Procedure Body

All procedures, unless standard, must be declared in the program in which they are called. In particular, when a program references a separately compiled procedure, the program must contain a declaration of that procedure. This declaration simply consists of a procedure heading and a code procedure body.

The procedure heading has the same format as a normal procedure heading (Section 5.4.1), and the code procedure body (defined as <code> in Section 5.4.1) consists of the symbol code followed by a number xxxxx in the range 0-99999. This is the same number associated with the procedure when it is compiled separately as an ALGOL source procedure (Chapter 4) and the compiler uses this number to link the declaration with the procedure. The procedure is linked to the main program at the object-program level. The object code does not necessarily have to be produced by the compilation of an ALGOL source procedure; it may be generated in any way provided that it conforms to the object code produced by the compilation of an ALGOL source procedure.

The identifying name included in the procedure heading need not be the same as the name declared in a separately compiled procedure, as linking is done by code number. However, all references in the program to the procedure must use the name declared in the program rather than the name declared in the separately declared procedure.

Names of formal parameters need not be the same as those declared when the procedure is compiled separately. Since the procedure may be capable of receiving a variable number of actual parameters, the number of formals in the procedure heading need not correspond with the number of actuals appearing in program references. In the calling program, the declaration containing a code procedure body need not have a value part of a specification part in the procedure heading.

The above rules also apply to a separately compiled procedure which references another separately compiled procedure. The referencing procedure must contain a declaration for the referenced procedure as described above.

In the following examples, the procedures AVERAGE and SQUAREAVERAGE may be compiled separately from the program in which they are referenced.

The following examples illustrate the use of separately compiled procedures. In the first example, the procedures AVERAGE and SQUAREAVERAGE are compiled in the original program.

Example 1.

```
begin
    real procedure AVERAGE (LOWER, UPPER);
        value LOWER, UPPER;
        real LOWER, UPPER;
        begin AVERAGE:=(LOWER+UPPER)/2;
        end;
    real procedure SQUAREAVERAGE (LOW, HIGH);
        value LOW, HIGH;
        real LOW, HIGH;
        begin SQUAREAVERAGE:=SQRT (LOW ↑2+HIGH ↑2)/2;
        end;
    real X, Y, S, SQ;
    S : = 0;
    SQ : = 0;
    for X : = 1 step 1 until 100 do
        begin
            Y : = X + 1;
            S : = S + AVERAGE (X, Y);
            SQ : = SQ + SQUAREAVERAGE (X, Y);
        end;
end
```

In the second example the first procedure body is replaced by the symbol code with the identifying number 129. In the heading, the identifying name AVERAGE has been changed to MEAN, and the formal parameter names to A and B. References to this procedure are to the name MEAN. The procedure called AVERAGE should be compiled separately with the code number 129.

The source deck (Chapter 4) for this compilation is:

```
code 129;
real procedure AVERAGE (LOWER, UPPER);
    value LOWER, UPPER;
    real LOWER, UPPER;
    begin AVERAGE : = (LOWER + UPPER)/2;
end;
```

followed by the 'EOP' indication in columns 10 through 14 of the next card.

The second procedure body is replaced by the symbol code and the identifying number 527. The procedure heading remains the same, except that the value and specification parts are omitted. Since the identifying name is not changed, the procedure is referenced as before. The procedure called SQUAREAVERAGE should be compiled separately with the code number 527.

Example 2.

```
begin
    real procedure MEAN (A, B);
        value A, B;
        real A, B;
        code 129;
    real procedure SQUAREAVERAGE (LOW, HIGH);
        code 527;
    real X, Y, S, SQ;
    S : = 0;
    SQ : = 0;
    for X : = 1 step 1 until 100 do
        begin
            Y : = X + 1;
            S : = S + MEAN (X, Y);
            SQ : = SQ + SQUAREAVERAGE (X, Y);
    end;
end
```

Examples of Procedure Declarations:

Example 1.

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;

integer tim ; real procedure fct ; real sum, eps ;

comment euler computes the sum of fct(i) for i from zero up to infinity by means of a suitably refined
euler transformation. The summation is stopped as soon as tim times in succession the absolute value of the
terms of the transformed series are found to be less than eps. Hence, one should provide a function fct with
one integer argument, an upper bound eps, and an integer tim. The output is the sum sum. euler is particularly
efficient in the case of a slowly convergent or divergent alternating series ;

begin integer i,k,n,t ; array m [0:15] ; real mn,mp,ds ;

i : = n := t := 0 ; m [0] : =fct(0) ; sum : = m [0] /2 ;

nextterm:i: =i+1 ; mn : =fct(i) ;

    for k : = 0 step 1 until n do

        begin mp : = (mn+m [k])/2 ; m [k] : = mn ;

        mn : = mp end means ;

    if (abs(mn)<abs(m [n] )) ∧ (n<15) then

        begin ds : = mn/2 ; n :=n+1 ; m[n] :=

        mn end accept

    else ds := mn ;

    sum : = sum + ds ;

    if abs(ds)<eps then t : =t+1 else t : = 0 ;

    if t<tim then go to nextterm

end euler
```

Example 2.†

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi) ; value x,y

integer n ; Boolean fi ; real x,eps,eta,xE ; array

y,yE ; procedure FKT ;
```

†This RK-program contains some new ideas which are related to ideas of S. Gill, A process for the step-by-step integration of differential equations in an automatic computing machine, [Proc. Camb. Phil. Soc. 47 (1951), 96] ; and E. Froberg, On the solution of ordinary differential equations with digital computing machines, [Fysiograf. Sällsk; Lund, Förhd. 20,11 (1950), 136-152] . It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

```
comment: RK integrates the system Yk'= fk(x,y1y2, . . . ,yn) (k = 1,2, . . . ,n) of differential equations with the
method of Runge-Kutta with automatic search for appropriate length of integration step. Parameters are: The
initial values x and y [k] for x and the unknown functions yk(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e. the set of functions fk. The tolerance values eps
and eta which govern the accuracy of the numerical integration. The end of the integration internal xE. The output
parameter yE which represents the solution at x = xE. The Boolean variable fi, which must always be given the
value true for an isolated or first entry into RK. If however the functions y must be available at several meshpoints
x0,x1, . . . ,xn, then the procedure must be called repeatedly (with x=xk,xE=xk+1, for k=0,1, . . . ,n-1) and then
the later calls may occur with fi=false which saves computing time. The input parameters of FKT must be x,y,n,
the output parameter z represents the set of derivatives z[k] =fk(x,y[1] ,y [2] , . . . ,y[n]) for x and the actual y's.
A procedure comp enters as a nonlocal identifier ;

begin

array z,y1,y2,y3 [1:n] ; real x1,x2,x3,H ; Boolean out ;

integer k,j ; own real s,Hs ;

procedure RK1ST (x,y,h,xe,ye) ;real x,h,xe ;array

        y,ye ;

comment: RK1ST integrates one single RUNGE-KUTTA with initial values x,y[k] which yields the output param-
eters xe=x+h and ye[k] , the latter being the solution at xe. Important: the parameters n, FKT, z enter RK1ST as
nonlocal entities ;

begin

array w[1:n] , a[1:5] ; integer k,j ;

a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;

xe :=x ;

for k : = 1 step 1 until n do ye[k] := w[k] :=y[k] ;

for j:= 1 step 1 until 4 do

begin

    FKT(xe,w,n,z) ;

    xe :=x+a[j] ;

    for k : = 1 step 1 until n do
```

```
    begin

        w[k] : = y[k] +a[j] × z[k]  ;

        ye[k]  := ye[k]  + a[j+1] × z[k] /3

            end k

        end j

    end RK1ST  ;

Begin of program:

    if fi then begin H : = xE–x  ; s : = 0 end else H : = Hs  ;

    out : = false  ;

AA: if(x+2.01×H–xE>0)≡(H>0) then

    begin Hs : = H  ;  out : = true  ; H : = (xE–x)/2

    end if  ;

    RK1ST (x,y,2×H,x1,y1)  ;

BB: RK1ST (x,y,H,x2,y2)  ;  RK1ST (x2,y2,H,x3,y3)  ;

    for k : = 1 step 1 until n do

        if comp(y1[k] ,y3[k] ,eta)>eps then go to CC  ;

    comment: comp(a,bc,) is a function designator, the value of which is the absolute value of the difference of the
    mantissae of a and b, after the exponents of these quantities have been made equal to the largest of the exponents
    of the originally given parameters a,b,c  ;

    x : = x3  ; if out then go to DD  ;

    for k : = 1 step 1 until n do y[k]  : = y3[k]  ;

    if s=5 then begin s : = 0  ; H : = 2×H end if  ;

    s : = s+1 ; go to AA  ;
```

```
CC: H : = 0.5×H  ; out : = false  ;x1 : = x2  ;

    for k : = 1 step 1 until n do y1[k] : = y2[k]  ;

    go to BB  ;

DD: for k:=1 step 1 until n do yE[k] : = y3[k]

end RK
```

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

- def — Following the abbreviation "def", reference to the syntactic definition (if any) is given.
- synt — Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.
- text — Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words have been collected at the beginning.

The examples have been ignored in compiling the index.

+, see: plus
−, see: minus
×, see: multiply
/, ÷, see: divide
↑, see: exponentiation
<, ≤, =, ≥, >, ≠, see: <relational operator>
≡, ⊃, ∨, ∧, ¬, see: <logical operator>
,, see: comma
., see: decimal point
$_{10}$, see: ten
:, see: colon
;, see: semicolon
:=, see: colon equal
␣, see: space
( ), see: parentheses
[ ], see: subscript brackets
‘ ’, see: string quotes

< actual parameter >, def 3.2.1, 4.7.1
< actual parameter list >, def 3.2.1, 4.7.1
< actual parameter part >, def 3.2.1, 4.7.1
< adding operator >, def 3.3.1
  alphabet, text 2.1
  arithmetic, text 3.3.6
< arithmetic expression >, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
< arithmetic operator >, def 2.3 text 3.3.4
  array, synt 2.3, 5.2.1, 5.4.1
  array, text 3.1.4.1
< array declaration >, def 5.2.1 synt 5 text 5.2.3
< array identifier >, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
< array list >, def 5.2.1
< array segment >, def 5.2.1
< assignment statement >, def 4.2.1 synt 4.1.1 text 1, 4.2.3

< basic statement >, def 4.1.1 synt 4.5.1
< basic symbol >, def 2
  begin, synt 2.3, 4.1.1
< block >, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
< block head >, def 4.1.1
  Boolean, synt 2.3, 5.1.1 text 5.1.3
< Boolean expression >, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
< Boolean factor >, def 3.4.1
< Boolean primary >, def 3.4.1
< Boolean secondary >, def 3.4.1
< Boolean term >, def 3.4.1
< bound pair >, def 5.2.1
< bound pair list >, def 5.2.1
< bracket >, def 2.3

< code >, synt 5.4.1 text 4.7.8, 5.4.6
  colon : , synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
  colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
  comma , , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1

comment, synt 2.3
comment convention, text 2.3
< compound statement >, def 4.1.1 synt 4.5.1 text 1
< compound tail >, def 4.1.1
< conditional statement >, def 4.5.1 synt 4.1.1 text 4.5.3

< decimal fraction >, def 2.5.1
< decimal number >, def 2.5.1 text 2.5.3
decimal point . , synt 2.3, 2.5.1 ,
< declaration >, def 5 synt 4.1.1 text 1, 5 (complete section)
< declarator >, def 2.3
< delimiter >, def 2.3 synt 2
< designational expression >, def 3.5.1 synt 3, 4.3.1., 5.3.1 text 3.5.3
< digit >, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
< dummy statement >, def 4.4.1 synt 4.1.1 text 4.4.3

else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2
< empty >, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
end, synt 2.3, 4.1.1
entier, text 3.2.5
exponentiation ↑ , synt 2.3, 3.3.1 text 3.3.4.3
< exponent part >, def 2.5.1 text 2.5.3
< expression >, def 3 synt 3.2.1, 4.7.1 text 3 (complete section)

< factor >, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
< for clause >, def 4.6.1 text 4.6.3
< for list >, def 4.6.1 text 4.6.4
< for list element >, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
< formal parameter >, def 5.4.1 text 5.4.3
< formal parameter list >, def 5.4.1
< formal parameter part >, def 5.4.1
< for statement >, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
< function designator >, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4

go to, synt 2.3, 4.3.1
< go to statement >, def 4.3.1 synt 4.1.1 text 4.3.3

< identifier >, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
< identifier list >, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
< if clause >, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
< if statement >, def 4.5.1 text 4.5.3.1
< implication >, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
< integer >, def 2.5.1 text 2.5.4

label, synt 2.3, 5.4.1
< label >, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3
< left part >, def 4.2.1
< left part list >, def 4.2.1
< letter >, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
< letter string >, def 3.2.1, 4.7.1
local, text 4.1.3
< local or own type >, def 5.1.1 synt 5.2.1
< logical operator >, def 2.3 synt 3.4.1 text 3.4.5
< logical value >, def 2.2.2 synt 2, 3.4.1
< lower bound >, def 5.2.1 text 5.2.4

minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
< multiplying operator >, def 3.3.1

nonlocal, text 4.1.3
< number >, def 2.5.1 text 2.5.3, 2.5.4

< open string >, def 2.6.1
< operator >, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5

< parameter delimiter >, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
parentheses ( ), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
< primary >, def 3.3.1
procedure, synt 2.3, 5.4.1
< procedure body >, def 5.4.1
< procedure declaration >, def 5.4.1 synt 5 text 5.4.3
< procedure heading >, def 5.4.1 text 5.4.3
< procedure identifier >, def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
< procedure statement >, def 4.7.1 synt 4.1.1 text 4.7.3
< program >, def 4.1.1 text 1
< proper string >, def 2.6.1

quantity, text 2.7

real, synt 2.3, 5.1.1 text 5.1.3
< relation >, def 3.4.1 text 3.4.5
< relational operator >, def 2.3, 3.4.1

scope, text 2.7
semicolon ; , synt 2.3, 4.1.1, 5.4.1
< separator >, def 2.3
< sequential operator >, def 2.3
< simple arithmetic expression >, def 3.3.1 text 3.3.3
< simple Boolean >, def 3.4.1
< simple designational expression >, def 3.5.1
< simple variable >, def 3.1.1 synt 5.1.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
< specification part >, def 5.4.1 text 5.4.5
< specificator >, def 2.3
< specifier >, def 5.4.1
standard function, text 3.2.4, 3.2.5
< statement >, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)

statement bracket, see: begin end
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
< string >, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ' ', synt 2.3, 2.6.1, text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript brackets [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
< subscripted variable >, def 3.1.1 text 3.1.4.1
< subscript expression >, def 3.1.1 synt 3.5.1
< subscript list >, def 3.1.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
< switch declaration >, def 5.3.1 synt 5 text 5.3.3
< switch designator >, def 3.5.1 text 3.5.3
< switch identifier >, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
< switch list >, def 5.3.1

< term >, def 3.3.1
ten $_{10}$, synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
< type >, def 5.1.1 synt 5.4.1 text 2.8
< type declaration >, def 5.1.1 synt 5 text 5.1.3
< type list >, def 5.1.1

< unconditional statement >, def 4.1.1, 4.5.1
< unlabelled basic statement >, def 4.1.1
< unlabelled block >, def 4.1.1
< unlabelled compound >, def 4.1.1
< unsigned integer >, def 2.5.1, 3.5.1
< unsigned number >, def 2.5.1 synt 3.3.1
until, synt 2.3, 4.6.1 text 4.6.4.2
< upper bound >, def 5.2.1 text 5.2.4

value, synt 2.3, 5.4.1
value, text 2.8, 3.3.3
< value part >, def 5.4.1 text 4.7.3.1
< variable >, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1
text 3.1.3
< variable identifier >, def 3.1.1

while, synt 2.3, 4.6.1 text 4.6.4.3

END OF THE REPORT

INPUT-OUTPUT 3

The processes of input and output deal with the mapping of basic characters onto input and output devices under the control of format rules. Characters are grouped to form lines, and lines are grouped to form pages. A page consists of printed lines and a line may be a printed line or card image.

The relation between lines and pages and physical entities (such as records and blocks) depends on formatting rules, channel specifications (B.1.1), SCOPE input-output, and the physical device involved in the input-output process. The user need not, in general, be aware of the details of this relationship, since the input-output process is symmetric. Given the same specifications, a file output by the system is disassembled into the same lines and pages as input by the system.

3.1 COMPARISON WITH ACM PROPOSAL FOR INPUT-OUTPUT

The following descriptions explain the differences between the input-output procedures included in ALGOL and the procedures defined in the ACM proposal.† To facilitate cross referencing, the same numbering system is used in this chapter as in the proposal. The ACM proposal is a continuation of the ALGOL-60 Revised Report, and should be considered a continuation of Chapter 2 of this manual.

All descriptions of the modifications to the input-output procedures are made at the main reference in the proposal; and wherever feasible, all other references are noted. The reader should assume, however, that such modifications apply to all references to the features, noted or otherwise.

A section or feature not mentioned in this chapter is implemented, in this version of ALGOL, in exact accordance with the proposal.

This chapter also contains descriptions of additional input-output procedures which are not defined in the ACM proposal, and a description of the transmission error, end-of-file, and end-of-tape functions automatically supplied within the framework of the input-output procedures.

† "A Proposal for Input-Output Conventions in ALGOL-60", published in The Communications of the ACM, vol. 7 no. 5, May 1964.

A Proposal for Input-Output Conventions in ALGOL-60

A Report of the Subcommittee on ALGOL of the ACM Programming Languages Committee

D. E. Knuth, Chairman
L. L. Bumgarner P. Z. Ingerman J. N. Merner
D. E. Hamilton M. P. Lietzke D. T. Ross

The ALGOL-60 language as first defined made no explicit reference to input and output processes. Such processes appeared to be quite dependent on the computer used, and so it was difficult to obtain agreement on those matters. As time has passed, a great many ALGOL compilers have come into use, and each compiler has incorporated some input-output facilities. Experience has shown that such facilities can be introduced in a manner which is compatible and consistent with the ALGOL language, and which (more importantly) is almost completely machine-independent. However, the existing implementations have taken many different approaches to the subject, and this has hampered the interchange of programs between installations. The ACM ALGOL committee has carefully studied the various proposals in an attempt to define a set of conventions for doing input and output which would be suitable for use on most computers. The present report constitutes the recommendations of that committee.

The input-output conventions described here do not involve extensions or changes to the ALGOL-60 language. Hence they can be incorporated into existing processors with a minimum of effort. The conventions take the form of a set of procedures,[1] which are to be written in code for the various machines; this report discusses the function and use of these procedures. The material contained in this proposal is intended to supplement the procedures in real, out real, in symbol, out symbol which have been defined by the international ALGOL committee; the procedures described here could, with trivial exceptions, be expressed in terms of these four.[2]

The first part of this report describes the methods by which formats are represented; then the calls on the input and output procedures themselves are discussed. The primary objective of the present report is to describe the proposal concisely and precisely, rather than to give a programmer's introduction to the input-output conventions. A simpler and more intuitive (but less exact) description can be written to serve as a teaching tool.

Many useful ideas were suggested by input-output conventions of the compilers listed in the references below. We are also grateful for the extremely helpful contributions of F. L. Bauer, M. Paul, H. Rutishauser, K. Samelson, G. Seegmuller, W. L. v.d. Poel, and other members of the European computing community, as well as A. Evans, Jr., R. W. Floyd, A. G. Grace, J. Green, G. E. Haynam, and W. C. Lynch of the USA.

A. Formats

In this section a certain type of string, which specifies the format of quantities to be input or output, is defined, and its meaning is explained.

A.1 Number Formats (cf. ALGOL Report 2.5)

A.1.1 Syntax

Basic components:

< replicator > :: = < unsigned integer > | X

< insertion > :: = B | < replicator > B | < string >

1 Throughout this report, names of system procedures are in lower case, and names of procedures used in illustrative examples are in upper case. (NOTE: Additional input-output procedures provided by CONTROL DATA are in upper case.)

2 Defined at meeting IFIP/WG2.1 – ALGOL in Delft during September, 1963.

< insertion sequence > :: = < empty >| < insertion sequence > < insertion >

< Z > :: = Z| < replicator > Z| Z < insertion sequence > C| < replicator >

Z < insertion sequence > C

< Z part > :: = < Z >| < Z part > < Z >| < Z part > < insertion >

< D > :: = D | < replicator > D| D < insertion sequence > C|

< replicator > D < insertion sequence > C

< D part > :: = < D >| < D part > < D >| < D part > < insertion >

< T part > :: = < empty >| T < insertion sequence >

< sign part > :: = < empty >| < insertion sequence > + |

< insertion sequence > -

< integer part > :: = < Z part >| < D part >| < Z part > < D part >

Format Structures:

< unsigned integer format > :: = < insertion sequence > < integer part >

< decimal fraction format > :: = . < insertion sequence > < D part >

< T part >| V < insertion sequence > < D part > < T part >

< exponent part format > :: = $_{10}$ < sign part > < unsigned integer format >

< decimal number format > :: = < unsigned integer format > < T part >|

< insertion sequence > < decimal fraction format >|

< unsigned integer format > < decimal fraction format >

< number format > :: = < sign part > < decimal number format >|

< decimal number format > + < insertion sequence >|

< decimal number format > - < insertion sequence >|

< sign part > < decimal number format > < exponent part format >

Note. This syntax could have been described more simply, but the rather awkward constructions here have been formulated so that no syntactic ambiguities (in the sense of formal language theory) will exist.

A.1.2 Examples. Examples of number formats appear in Figure 1.

The letter C is not implemented. All references to C in the ACM Report should be disregarded, e.g., <Z> : : = Z | <replicator> Z | Z <insertion sequence> C | <replicator> Z <insertion sequence> C will be implemented as follows:

<Z> : : = Z | <replicator> Z

Number format	Result from −13.296	Result from 1007.999
+ZZZCDDD.DD	−013.30	+1,008.00
+3ZC3D.2D	−013.30	+1,008.00
−3D2B3D.2DT	−000 013.29	001 007.99
5Z.5D−	13.29600−	1007.99900
'integer␣part␣'−4ZV	integer part −13,	integer part 1007,
',␣fraction'B3D	fraction 296	fraction 999
−.5D$_{10}$+2D'. . .'	−.13296$_{10}$+02. . .	.100801$_{0}$+04. . .
+ZD$_{10}$2Z	−13	+10$_{10}$ 2
+D.DDBDDBDDB$_{10}$ +DD	−1.32 96 00 10+01	+1.00 79 99 $_{10}$ +03
XB.XD $_{10}$−DDD	(depends on call)	(depends on call)

Figure 1

Figure 1 (depends on call) - for definition of call see A.1.3.3 Sign and Zero Suppression.

A. Formats

A format string may be a string or an array into which a string has been read, using H format (Section A.2.3.3). When a format string can be specified either form can be used.

A.1.3 Semantics. The above syntax defines the allowable strings which can comprise a "number format." We will first describe the interpretation to be taken during output.

A.1.3.1 Replicators. An unsigned integer n used as replicator means the quantity is repeated n times; thus 3B is equivato BBB. The character X as replicator means a number of times which will be specified when the format is called (see Section B.3.1).

A.1.3.1 Replicators

A replicator of value 0 (n or X) implies the absence of the quantity to which the replicator refers. The maximum size of a replicator is 262,142.

A.1.3.2 Insertions. The syntax has been set up so that strings, delimited by string quotes, may be inserted anywhere within a number format. The corresponding information in the strings (except for the outermost string quotes) will appear inserted in the same place with respect to the rest of the number. Similarly, the letter B may be inserted anywhere within a number format, and it stands for a blank space.

A.1.3.3 Sign, zero, and comma suppression. The portion of a number to the left of the decimal point consists of an optional sign, then a sequence of Z's and a sequence of D's with possible C's following a Z or a D, plus possible insertion characters.

The convention on signs is the following: (a) if no sign appears, the number is assumed to be positive, and the treatment of negative numbers is undefined: (b) if a plus sign appears, the sign will appear as + or – on the external medium; and (c) if a minus sign appears, the sign will appear if minus, and will be suppressed if plus.

The letter Z stands for zero suppression, and the letter D stands for digit printing without zero suppression. Each Z and D stands for a single digit position; a zero digit specified by Z will be suppressed, i.e., replaced by a blank space, when all digits to its left are zero. A digit specified by D will always be printed. Note that the number zero printed with all Z's in the format will give rise to all blank spaces, so at least one D should usually be given somewhere in the format. The letter C stands for a comma. A comma following a D will always be printed; a comma following a Z will be printed except when zero suppression takes place at that Z. Whenever zero or comma suppression takes place, the sign (if any) is printed in place of the rightmost character suppressed.

A.1.3.3 Sign and Zero Suppression

On input, if no sign appears in the format and the number is negative, an error condition exists. If the procedure BAD DATA (Section 3.3) has not been called, or if the label established by it is no longer accessible, an error message is issued and the object program terminates abnormally (Chapter 8). Otherwise, control is transferred to the BAD DATA label. Output uses the standard format bounded on either side by an asterisk (Section A.2.3.6). Comma suppression is not implemented.

A.1.3.4 Decimal points. The position of the decimal point is indicated either by the character "." or by the letter V. In the former case, the decimal point appears on the external medium; in the latter case, the decimal point is "implied" i.e., it takes up no space on the external medium. (This feature is most commonly used to save time and space when preparing input data). Only D's (no Z's) may appear to the right of the decimal point.

A.1.3.4 Decimal points. In an exponent part, D's and Z's may appear to the right of the decimal point.

A.1.3.5 Truncation. On output, nonintegral numbers are usually rounded to fit the format specified. If the letter T is used, however, truncation takes place instead. Rounding and truncation of a number X to d decimal places are defined as follows:

Rounding $10^{-d}\text{entier}(10^{d}X + 0.5)$

Truncation $10^{-d}\text{sign}(X)\ \text{entier}\ (10^{d}\ \text{abs}(X))$

A.1.3.5 Truncation

On output, the number of significant digits appearing for a real number will correspond to the storage of the number in 60-bit floating-point form. Thus 14 or 15 significant digits are output, followed by trailing zeros, if necessary (Section 5.1.3). The letter T has no meaning when applied to an integer number and is ignored.

A.1.3.6 Exponent part. The number following a "$_{10}$" is treated exactly the same as the portion of a number to the left of a decimal point (Section A.1.3.3), except if the "D" part of the exponent is empty, i.e., no D's appear, and if the exponent is zero, the "$_{10}$" and the sign are deleted.

A.1.3.7 Two types of numeric format. Number formats are of two principal kinds: (a) Decimal number with no exponent. In this case, the number is aligned according to the decimal point with the picture in the format, and it is then truncated or rounded to the appropriate number of decimal places. The sign may precede or follow the number.

(b) Decimal number with exponent. In this case, the number is transformed into the format of the decimal number with its most significant digit non-zero; the exponent is adjusted accordingly. If the number is zero, both the decimal part and the exponent part are output as zero. If in case (a) the number is too large to be output in the specified form, or if in case (b) the exponent is too large, an overflow error occurs. The action which takes place on overflow is undefined; it is recommended that the number of characters used in the output be the same as if no overflow had occurred, and that as much significant information as possible be output.

A.1.3.7 Two Types of Numeric Format

A maximum of 24 D's and Z's may appear before the exponent part in a number format; in the exponent part, the maximum is 4. On output overflow, the standard format bounded on either side by an asterisk is used (Section A.2.3.6).

A.1.3.8 Input. A number input with a particular format specification should in general be the same as the number which would be output with the same format, except less error checking occurs. The rules are, more precisely:

(a) leading zeros and commas may appear even though Z's are used in the format. Leading spaces may appear even if D's are used. In other words, no distinction between Z and D is made on input.

(b) Insertions take the same amount of space in the same positions, but the characters appearing there are ignored on input. In other words, an insertion specifies only the number of characters to ignore, when it appears in an input format.

(c) If the format specifies a sign at the left, the sign may appear in any Z,D or C position as long as it is to the left of the number. A sign specified at the right must appear in place.

(d) The following things are checked: The positions of commas, decimal points, "$_{10}$" and the presence of digits in place of D or Z after the first significant digit. If an error is detected in the data, the result is undefined; it is recommended that the input procedure attempt to reread the data as if it were in standard format (Section A.5) and also to give some error indication compatible with the system being used. Such an error indication might be suppressed at the programmer's option if the data became meaningful when it was reread in standard format.

A.1.3.8 Input

If the input data does not conform to the format, an error condition exists. If the procedure BAD DATA was not called or if the established label is no longer accessible, an error message is issued; and the object program terminates abnormally. Otherwise, control is transferred to the BAD DATA label. C is not implemented.

A.2 Other formats

A.2.1 Syntax

< S > :: = S| < replicator > S

< string format > :: = < insertion sequence > < S >| < string format > < S >| < string format > < insertion >

< A > :: = A| < replicator > A

< alpha format > :: = < insertion sequence > < A >| < alpha format > < A >|

< alpha format > < insertion >

< nonformat > :: = I| R| L

< Boolean part > :: = P| 5F| FFFFF| F

< Boolean format > :: = < insertion sequence > < Boolean part >

< insertion sequence >

< title format > :: = < insertion >| < title format > < insertion >

< alignment mark > :: = / |↑ | < replicator > /| < replicator > ↑

< format item 1 > :: = < number format >| < string format >|

< alpha format >|< nonformat >|< Boolean format >|< title format >|

< alignment mark > < format item 1 >

< format item > :: = < format item 1 >| < alignment mark >| < format item > < alignment mark >

A.2.2 Examples

```
↑5Z.5D///
3S'='6S4B
AA'='
↑R
P
/'Execution.'↑
```

The following definitions replace the definitions in the ACM Report:

A. 2 Other Formats

A. 2. 1 Syntax

<S> : : = S | <replicator> S

<string format> : : = <insertion sequence> <S> | <string format>

<S> | <string format> <insertion>

<A> = A | <replicator>A

<alpha format> : : = <insertion sequence> <A> | <alpha format> <A> |

<alpha format> <insertion>

<standard format> : : = N

<nonformat> : : = I | R | L | M | H

<Boolean part> : : = P | F

<Boolean format> : : = <insertion sequence> <Boolean part> <insertion sequence>

<title format> : : = <insertion> |

<title format> <insertion>

<alignment mark> : : = / | ↑ | J |

<replicator> / | <replicator> ↑ |

<replicator> J

<format item 1> : : = <number format> |

<string format> | <alpha format>

| <nonformat> | <Boolean format> |

<title format> | <alignment mark>

<format item 1> | <standard format>

<format item> : : = <format item 1> |

<alignment mark> | <format item>

<alignment mark>

The characters M and H have been added to the non-format codes.

J has been added to alignment mark (Section B. 3).

A.2.3 Semantics

A.2.3.1 String format. A string format is used for output of string quantities. Each of the S-positions in the format corresponds to a single character in the string which is output. If the string is longer than the number of S's, the leftmost characters are transferred; if the string is shorter, ␣-symbols are effectively added at the right of the string.

The word "character" as used in this report refers to one unit of information on the external input or output medium; if ALGOL basic symbols are used in strings which do not have a single-character representation on the external medium being used, the result is undefined.

A. 2. 3 Semantics

The maximum length of a format item, after expanding each quantity in it by the corresponding replicator, is 136 characters; the expanded format item corresponds to the data on the external device.

A.2.3.1 String Format

Because of the difference in the definition of a string (Section 2.6.1), each of the S-positions in the format corresponds to a single basic character in the output string rather than a single basic symbol. If the string exceeds the number of S's, the leftmost basic characters are transferred; if the string is shorter, blank characters are filled to the right.

A.2.3.2 Alpha format. Each letter A means one character is to be transmitted; this is the same as S-format except the ALGOL equivalent of the alphabetics is of type integer rather than a string. The translation between the external and internal code will vary from one machine to another, and so programmers should refrain from using this feature in a machine-dependent manner. Each implementor should specify the maximum number of characters which can be used for a single integer variable. The following operations are undefined for quantities which have been input using alpha format; arithmetic operations, relations except "=" and "≠", and output using a different number of A's in the output format. If the integer is output using the same number of A's, the same string will be output as was input.

A programmer may work with these alphabetic quantities in a machine-independent manner by using the transfer function equiv(S) where S is a string; the value of equiv(S) is of type integer, and it is defined to have exactly the same value as if the string S had been input using alpha format. For example, one may write

if X = equiv('ALPHA') then go to PROCESS ALPHA;

where the value of X has been input using the format "AAAAA".

A.2.3.2 Alpha Format

Because of the difference in the definition of a string (Section 2.6.1), each letter A indicates the transmission of one basic character rather than one basic symbol. The maximum number of characters which may be used for a single integer variable is defined as 8. Thus the maximum replication of A within a format item is 8. The A format is the same as S format except that the ALGOL equivalent of the basic character is of type integer rather than string.

Similarly, the transfer function EQUIV (S) is an integer procedure, the value of which is the internal representation of the first 8 basic characters in the string S. If the string contains less than 8 characters, blanks are added to the right. Thus the value of EQUIV (S) is the same as if the string characters had been input under A format. The EQUIV function (as well as input under A and H formats) assigns an internal representation of zero to a blank character to facilitate limited string manipulation by integer arithmetic, as leading blanks are not significant. In the 64 character set, use of zero for blank means that the character normally assigned to display code zero (colon) cannot be input or output under A or H format. It can still be input and output under S format or standard format.

A.2.3.3 Nonformat. An I, R or L is used to indicate that the value of a single variable of integer, real, or Boolean type, respectively, is to be input or output from or to an external medium, using the internal machine representation. If a value of type integer is output with R-format or if a value of type real is input with I-format, the appropriate transfer function is invoked. The precise behavior of this format, and particularly its interaction with other formats, is undefined in general.

A.2.3.3 Nonformat

The M code added to the nonformat codes indicates that the value of a single variable of any type is to be input or output in the exact form in which it appears on the external device or in memory.

The nonformat codes I, R, L and M each input or output 20 consecutive (6-bit) display characters, and map them to or from the 20 consecutive octal (3-bit) digits which constitute one variable internally (Section 5.1.3).

The H code added to the nonformat codes indicates that 8 consecutive display characters are to be input or output to or from a single integer variable.

A.2.3.4 Boolean format. When Boolean quantities are input or output, the format P, F, 5F or FFFFF must be used. The correspondence is defined as follows:

Internal to ALGOL	**P**	**F**	**5F = FFFFF**
true	**1**	**T**	**TRUE␣**
false	**0**	**F**	**FALSE**

On input, anything failing to be in the proper form is undefined.

A.2.3.4 Boolean format. When Boolean quantities are input or output, the format P, F, must be used. The correspondence is defined as follows (Section A.2.1):

Internal to ALGOL	P	F
true	1	T
false	0	F

External representations in F format are t and f rather than true or false.

On input, incorrect forms cause an error condition. If the procedure BAD DATA was not called or if the established label is no longer accessible, an error message is issued and the object program terminates abnormally.

A.2.3.5 Title format. All formats discussed so far have given a correspondence between a single ALGOL real, integer, Boolean, or string quantity and a number of characters in the input or output. A title format item consists entirely of insertions and alignment marks, and so it does not require a corresponding ALGOL quantity. On input, it merely causes skipping of the characters, and on output it causes emission of the insertion characters it contains. (If titles are to be input, alpha format should be used; see Section A.2.3.2).

A.2.3.6 Alignment marks. The characters "/" and "↑" in a format item indicate line and page control actions. The precise definition of these actions will be given later (see Section B.5); they have the following intuitive interpretation: (a) "/" means go to the next line, in a manner similar to the "carriage return" operation on a typewriter. (b) "↑" means do a /-operation and then skip to the top of the next page.

Two or more alignment marks indicates the number of times the operations are to be performed; for example, "//" on output means the current line is completed and the next line is effectively set to all blanks. Alignment marks at the left of a format item cause actions to take place before the regular format operation, and if they are at the right they take place afterwards.

Note. On machines which do not have the character ↑ in their character set, it is recommended that some convenient character such as an asterisk be substituted for ↑ in format strings.

A. 2. 3. 6 Alignment marks. The character J has been added to indicate character control action in a format item. J means skip the character pointer to the next tabulation position; similar to the tab operation on a typewriter.

An asterisk may be substituted for ↑ in format strings.

A.3 Format Strings

The format items mentioned above are combined into format strings according to the rules in this section.

A.3.1 Syntax

< format primary > :: = < format item >|

< replicator > (< format secondary >)|(< format secondary >)

< format secondary > :: = < format primary >|

< format secondary > , < format primary >

< format string > :: = ' < format secondary > '|' '

A.3.2 Examples

'4 (15ZD) ,//'
'↑'
'.5D_{10}+D,X(2(20B.8D_{10}+D) ,10S) '
". . . This ␣ is ␣ a ␣ peculiar ␣ 'format ␣ string' "

A.3.3 Semantics. A format string is simply a list of format items, which are to be interpreted from left to right. The construction " < replicator > (< format secondary >) " is simply an abbreviation for "replicator" repetitions of the parenthesized quantity (see Section A.1.3.1). The construction " (< format secondary >) " is used to specify an infinite repetition of the parenthesized quantity.

All spaces within a format string except those which are part of insertion substrings are irrelevant.

It is recommended that the ALGOL compiler check the syntax of strings which (from their context) are known to be format strings as the program is compiled. In most cases it will also be possible for the compiler to translate format strings into an intermediate code designed for highly efficient input-output processing by the other procedures.

A. 3. 3 Semantics

The infinite repetition of the parenthesized quantity is defined as meaning 262,142 repetitions.

A.4 Summary of Format Codes

A	alphabetic character represented as integer	X	arbitrary replicator
B	blank space	Z	zero suppression
C	comma	+	print the sign
D	digit	–	print the sign if it is minus
F	Boolean TRUE or FALSE	10	exponent part indicator
I	integer untranslated	( )	delimiters of replicated format secondaries
L	Boolean untranslated	,	separates format items
P	Boolean bit	/	line alignment
R	real untranslated	↑	page alignment
S	string character	' '	delimiters of inserted string
T	truncation	.	decimal point
V	implied decimal point		

The following items have been added to format codes:

J	character alignment
N	standard format
M	variable of any type
H	integer variable;

and C has been deleted.

A.5 "Standard" Format

There is a format available without specifications (cf. Section B.5) which has the following characteristics.

(a) On input, any number written according to the ALGOL syntax for < number > is accepted with the conventional meaning. These are of arbitrary length, and they are delimited at the right by the following conventions: (i) A letter or character other than a decimal point, sign, digit, or "10" is a delimiter. (ii) A sequence of k or more blank spaces serves as a delimiter as in (i); a sequence of less than k blank spaces is ignored. This number $k \geqslant 1$ is specified by the implementor (and the implementor may choose to let the programmer specify k on a control card of some sort). (iii) If the number contains a decimal point, sign, digit, or "10" on the line where the number begins, the right-hand margin of that line serves as a delimiter of the number. However, if the first line of a field contains no such characters, the number is determined by reading several lines until finding a delimiter of type (i) or (ii). In other words, a number is not usually split across more than one line, unless its first line contains nothing but spaces or characters which do not enter into the number itself. (See Section B.5 for further discussion of standard input format.)

(b) On output, a number is given in the form of a decimal number with an exponent. This decimal number has the amount of significant figures which the machine can represent; it is suitable for reading by the standard input format. Standard output format takes a fixed number of characters on the output medium; this size is specified by each ALGOL installation. Standard output format can also be used for the output of strings, and in this case the number of characters is equal to the length of the string.

A. 5 Standard Format

Standard format may be invoked by the format item N, through the exhaustion of the format string, or by specifying an empty format string. The standard format of both integer and real variables is $+D.13D_{10}+3D$. When the given format is incorrect, the modified standard format is '*' $+D.13D_{10}+3D$'*' (Sections A.1.3.3 and A.1.3.7).

The number of blank characters, k, serving as a delimiter between numbers in standard format may be specified on the channel card (Chapter 7). If not specified, two is assumed.

String parameters can be output under standard format, nS, where n is the length of the string.

B. Input and Output Procedures

B.1 General Characteristics

The over-all approach to input and output which is provided by the procedures of this report will be introduced here by means of a few examples, and the precise definition of the procedures will be given later.

Consider first a typical case, in which we want to print a line containing the values of the integer variables N and M, each of which is nonnegative, with at most five digits; also the value of X[M], in the form of a signed number with a single nonzero digit to the left of the decimal point, and with an exponent indicated; and finally the value of cos(t), using a format with a fixed decimal point and no exponent. The following might be written for this case:

```
output 4(6,'2(BBBZZZZD) ,3B+D.DDDDDD10+DDD,3B,

-Z.DDDDBDDDD/',N,M,X[M] ,cos (t)).
```

This example has the following significance. (a) The "4" in output 4 means four values are being output. (b) The "6" means that output is to go to unit 6.

This is the logical unit number, i.e., the programmer's number for that unit, and it does not necessarily mean physical unit number 6. See Section B.1.1, for further discussion of unit numbers. (c) The next parameter,'2(BBB . . . DDDD/', is the format string which specifies a format for outputting the four values. (d) The last four parameters are the values being printed. If N = 500, M = 0, X[0] = 18061579, and t = 3.1415926536, we obtain the line

```
␣␣␣␣␣500␣␣␣␣␣␣␣␣0␣␣␣+1.806158 10+007␣␣␣-1.0000␣0000
```

as output.

Notice the "/" used in the above format; this symbol signifies the end of a line. If it had not been present, more numbers could have been placed on the same line in a future output statement. The programmer may build the contents of a line in several steps, as his algorithm proceeds, without automatically starting a new line each time output is called. For example, the above could have been written

```
output 1(6,'BBBZZZZD',N);
output 1(6,'BBBZZZZD',M);
```

```
output 2(6,'3B+D.DDDDDD,10+DDD,3B,-Z.DDDDBDDDD',
    X[M],cos(t)) ;
output 0(6,'/');
```

with equivalent results.

In the example above a line of 48 characters was output. If for some reason these output statements are used with a device incapable of printing 48 characters on a single line, the output would actually have been recorded on two or more lines, according to a rule which automatically keeps from breaking numbers between two consecutive lines wherever possible. (The exact rule appears in Section B.5)

Now let us go to a slightly more complicated example:

the real array A[1:n,1:n] is to be printed, starting on a new page. Supposing each element is printed with the format "BB-ZZZZ.DD", which uses ten characters per item, we could write the following program:

```
output 0(6,'↑');
for i := 1 step 1 until n do
begin for j := 1 step 1 until n do output 1(6,'BB-ZZZZ.DD',
    A[i,j]); output 0(6,'//')end.
```

If 10n characters will fit on one line, this little program will print n lines, double spaced, with n values per line; otherwise n groups of k lines separated by blank lines are produced, where k lines are necessary for the printing of n values. For example, if n = 10 and if the printer has 120 character positions, 10 double-spaced lines are produced. If, however, a 72-character printer is being used, 7 values are printed on the first line, 3 on the next, the third is blank, then 7 more values are printed, etc.

There is another way to achieve the above output and to obtain more control over the page format as well. The subject of page format will be discussed further in Section B.2, and we will indicate here the manner in which the above operation can be done conveniently using a single output statement. The procedures output 0, output 1, etc. mentioned above provide only for the common cases of output, and they are essentially a special abbreviation for certain calls on the more general procedure out list. This more general procedure could be used for the above problem in the following manner:

```
out list (6,LAYOUT,LIST)
```

Here LAYOUT and LIST are the names of procedures which appear below. The first parameter of out list is the logical unit number as described above. The second parameter is the name of a so-called "layout procedure"; general layout procedures are discussed in Section B.3. The third parameter of out list is the name of a so-called "list procedure"; general list procedures are discussed in Section B.4. In general, a layout procedure specifies the format control of the input or output. For the case we are considering, we could write a simple layout procedure (named "LAYOUT") as follows:

```
procedure LAYOUT; format 1('↑,(X(BB-ZZZZ.DD),//)',n)
```

The 1 in format 1 means a format string containing one X is given.

The format string is ↑,

```
(X(BB-ZZZZ.DD),//)
```

which means skip to a new page, then repeat the format X(BB–ZZZZ.DD),// until the last value is output. The latter format means that BB–ZZZZ.DD is to be used X times, then skip to a new line. Finally, format 1 is a procedure which effectively inserts the value of n for the letter X appearing in the format string.

A list procedure serves to specify a list of quantities. For the problem under consideration, we could write a simple list procedure (named "LIST") as follows:

```
procedure LIST(ITEM); for i := step 1 until n do
    for j := 1 step 1 until n do ITEM(A[i,j])
```

Here "ITEM A[i,j]" means that A[i,j] is the next item of the list. The procedure ITEM is a formal parameter which might have been given a different name such as PIECE or CHUNK; list procedures are discussed in more detail in Section B.4.

The declarations of LAYOUT and LIST above, together with the procedure statement out list(6,LAYOUT,LIST), accomplish the desired output of the array A.

Input is done in a manner dual to output, in such a way that it is the exact inverse of the output process wherever possible. The procedures in list and input n correspond to out list and output n (n = 0,1, . . .) . Two other procedures, get and put, are introduced to facilitate storage of intermediate data on external devices. For example, the statement put (100,LIST) would cause the values specified in the list procedure named LIST to be recorded in the external medium with an identification number of 100. The subsequent statement get (100,LIST) would restore these values. The external medium might be a disk file, a drum, a magnetic tape, etc.; the type of device and the format in which data is stored there is of no concern to the programmer.

B.1.1 Unit numbers. The first parameter of input and output procedures is the logical unit number, i.e., some number which the programmer has chosen to identify some input or output device. The connection between logical unit numbers and the actual physical unit numbers is specified by the programmer outside of the ALGOL language, by means of "control cards" preceding or following his program, or in some other way provided by the ALGOL implementor. The situation which arises if the same physical unit is being used for two different logical numbers, or if the same physical unit is used both for input and for output, is undefined in general.

It is recommended that the internal computer memory (e.g. the core memory) be available as an "input-output device", so that data may be edited by means of input and output statements.

B.1.1 Unit Numbers

Wherever the term unit number appears in the ACM Report, channel number applies. This channel number is synonymous with unit number in the ACM Report.

A channel is defined as all the specifications the I/O system needs to perform operations on a particular data file. A channel may be thought of as the set of descriptive information by which one reaches or knows of a data file. A channel number is the name of this set of descriptive information as well as the internal, indirect reference name of the data file accessed via this information.

The channel contains the following specifications about a data file:

1. Physical device description — device name, logical address, read or write mode

2. Status of physical device — binary or BCD, device position, error conditions

3. Data file description — file name, read or write mode, blocking information

4. Data file status — file position, error conditions

5. Description of formatting area — buffer area from or to which data in a file is moved

6. List of labels to which control will be given if errors occur.

Channels are established by means of channel cards (Chapter 7).

B.2 Horizontal and Vertical Control

This section deals with the way in which the sequence of characters, described by the rules of formats in Section A, is mapped onto input and output devices. This is done in a manner which is essentially independent of the device being used, in the sense that with these specifications the programmer can anticipate how the input or output data will appear on virtually any device. Some of the features of this description will, of course, be more appropriately used on certain devices than on others.

We will begin by assuming we are doing output to a printer. This is essentially the most difficult case to handle, and we will discuss the manner in which other devices fit into the same general framework. The page format is controlled by specifying the horizontal and the vertical layout. Horizontal layout is controlled essentially in the same manner as vertical layout, and this symmetry between the horizontal and vertical dimensions should be kept in mind for easier understanding of the concepts of this section.

Refer to Figure 2; the horizontal format is described in terms of three parameters (L,R,P), and the vertical format has corresponding parameters (L',R',P'). The parameters L, L' and R, R' indicate left and right margins, respectively; Figure 2 shows a case where L = L' = 4 and R = R' = 12. Only positions L through R of a horizontal line are used, and only lines L' and R' of the page are used; we require that $1 \leq L \leq R$ and $1 \leq L' \leq R'$. The parameter P is the number of characters per line, and P' is the number of lines per page. Although L, R, L' and R' are chosen by the programmer, the values of P and P' are characteristics of the device and they are usually out of the programmer's control. For those devices on which P and P' can vary (for example, some printers have two settings, one on which there are 66 lines per page, and another on which there are 88), the values are specified to the system in some manner external to the ALGOL program, e.g. on control cards. For certain devices, values P or P' might be essentially infinite.

B. 2 Horizontal and Vertical Control

The values are specified to the system by a suitable call on the procedure SYSPARAM (Section B. 6) or with channel cards.

The initial value of P on the channel card (Chapter 7) defines the maximum size of the line to be read or written. P may be changed during program execution, but it may never exceed its initial setting. The initial value of P' on the channel card defines the number of lines per page; the value of this parameter may be changed to exceed its initial setting.

Although Figure 2 shows a case where $P \geq R$ and $P' \geq R'$, it is of course quite possible that $P < R$ or $P' < R'$ (or both) might occur, since P and P' are in general unknown to the programmer. In such cases, the algorithm described in Section B.5 is used to break up logical lines which are too wide to fit on a physical line, and to break up logical pages which are too large to fit a physical page. On the other hand, the conditions $L \leq P$ and $L' \leq P'$ are insured by setting L or L' equal to 1 automatically if they happen to be greater than P or P', respectively.

Characters determined by the output values are put onto a horizontal line; there are three conditions which cause a transfer to the next line: (a) normal line alignment, specified by a "/" in the format; (b) R-overflow, which occurs when a group of characters is to be transmitted which would pass position R; and (c) P-overflow, which occurs when a group of characters is to be transmitted which would not cause R-overflow but would pass position P. When any of these three things occurs, control is transferred to a procedure specified by the programmer in case special action is desired (e.g. a change of margins in case of overflow; see Section B.3.3).

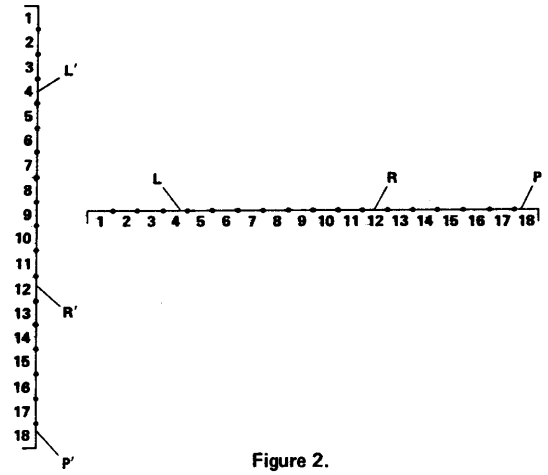


Figure 2.

Similarly, there are three conditions which cause a transfer to the next page: (a') normal page alignment, specified by a "↑" in the format; (b') R'-overflow, which occurs when a group of characters is to be transmitted which would appear on line R'+1; and (c') P'-overflow, which occurs when a group of characters is to be transmitted which would appear on line P'+1<R'+1. The programmer may indicate special procedures to be executed at this time if he wishes, e.g. to insert a page heading, etc.

Further details concerning pages and lines will be given later. Now we will consider how devices other than printers can be thought of in terms of the ideas above.

A typewriter is, of course, very much like a printer and it requires no further comment.

Punched cards with, say, 80 columns, have P = 80 and P' = ∞. Vertical control would appear to have little meaning for punched cards, although the implementor might choose to interpret "↑" to mean the insertion of a coded or blank card.

With paper tape, we might again say that vertical control has little or no meaning; in this case, P could be the number of characters read or written at a time.

On magnetic tape capable of writing arbitrarily long blocks, we have P = P' = ∞. We might think of each page as being a "record", i.e., an amount of contiguous information on the tape which is read or written at once. The lines are subdivisions of a record, and R' lines form a record; R characters are in each line. In this way we can specify so-called "blocking of record." Other interpretations might be more appropriate for magnetic tapes at certain installations, e.g. a format which would correspond exactly to printer format for future offline listing, etc.

These examples are given merely to indicate how the concepts described above for printers can be applied to other devices. Each implementor will decide what method is most appropriate for his particular devices, and if there are

choices to be made they can be given by the programmer by means of control cards. (channel cards) The manner in which this is done is of no concern in this report; our procedures are defined solely in terms of P and P'.

B.3 Layout Procedures

Whenever input or output is done, certain "standard" operations are assumed to take place, unless otherwise specified by the programmer. Therefore one of the parameters of the input or output procedure is a so-called "layout" procedure, which specifies all of the nonstandard operations desired. This is achieved by using any or all of the six "descriptive procedures" format, h end, v end, h lim, v lim, no data described in this section.

The precise action of these procedures can be described in terms of the mythical concept of six "hidden variables," H1, H2, H3, H4, H5, H6. The effect of each descriptive procedure is to set one of these variables to a certain value; and as a matter of fact, that may be regarded as the sum total of the effect of a descriptive procedure. The programmer normally has no other access to these hidden variables (see, however, Section B.7). The hidden variables have a scope which is local to in list and to out list.

B.3 A seventh descriptive procedure, TABULATION, has been added with its corresponding hidden variable H7 (Section B.3.3).

Tabulation is controlled by a J in the format. This causes the character pointer to be advanced to the next "TAB" position with intermediate positions being filled with blanks. The tabulation spacing for the device may be specified external to the ALGOL system, or through a suitable call on SYSPARAM (Section B.6).

If the tabulation spacing is N, then the first character of tabulation fields would be:

$$L,\ L+N,\ L+2N,\ \ldots,\ L+KN \text{ where } L+KN \leq \min(P, R).$$

If any of the procedures FORMAT, H END, V END, H LIM, V LIM, TABULATION, or NO DATA are called when neither IN LIST nor OUT LIST is active, they have the effect of a dummy procedure; a procedure call is made and the procedure is exited immediately.

B.3.1 Format Procedures. The descriptive procedure call

format (string)

has the effect of setting the hidden variable H1 to indicate the string parameter. This parameter may either be a string explicitly written, or a formal parameter; but in any event, the string it refers to must be a format string, which satisfies the syntax of Section A.3, and it must have no "X" replicators.

The procedure format is just one of a class of procedures which have the names format n, (n = 0, 1, . . .). The name format is equivalent to format 0. In general, the procedure format n is used with format strings which have exactly n X-replicators. The call is

format n (string, $X_1, X_2, \ldots X_n$)

where each X_i is an integer parameter called by value. The effect is to replace each X of the format string by one of the X_i, with the correspondence defined from left to right. Each X_i must be nonnegative.

For example,

format 2 ('XB . XD_{10}+DD', 5,10)

is equivalent to

format ('5B . 10D$_{10}$+DD').

B.3.1 Format Procedures

The single procedure with call:

FORMAT (string, X_1, X_2, . . . ,X_n)

replaces the n+1 procedures defined in the proposal with call:

FORMAT n (string, X_1, X_2, . . . ,X_n)

The number of X_i variables included in the call to FORMAT defines the equivalent n+1 procedure defined in the proposal. For example:

FORMAT (string, X_1, X_2) is equivalent to

FORMAT 2 (string, X_1, X_2) defined in the proposal.

A call to FORMAT may include 0-30 variables. The number depends on the parenthesized structure of the string.

B.3.2 Limits. The descriptive procedure call

h lim (L,R)

has the effect of setting the hidden variable H2 to indicate the two parameters L and R. Similarly,

v lim (L',R')

sets H3 to indicate L' and R'. These parameters have the significance described in Section B.2. If h lim and v lim are not used, L = L' = 1 and R = R' = ∞.

B.3.2 Limits

Since the first character of each record is used by the system to control skipping when paging is requested, the H LIM procedure increases the values of the L and R parameters by 1 to overcome the loss of this character. Any attempt to set the L and R parameters so that R-L ≤ 21 is ignored.

B.3.3 End Control. The descriptive procedure

h end (P_N,P_R,P_P); v end ($P_{N'}$,$P_{R'}$,$P_{P'}$);

have the effect of setting the hidden variables H4 and H5, respectively, to indicate their parameters. The parameters P_N,P_R,P_P,$P_{N'}$,$P_{R''}$;$P_{P'}$ are names of procedures (ordinally dummy statements if h end and v end are not specified) which are activated in case of normal line alignment, R-overflow, P-overflow, normal page alignment, R'-overflow, and P'-overflow, respectively.

B.3.3 The descriptive procedure call

TABULATION (n)

has the effect of setting the hidden variable H7 (Section B.3.5) to indicate the parameter n. Here n is the width of the tabulation field, measured in the number of characters on the external device (Section B.5.1, Process C). If tabulation is not called then H7 = 1.

B.3.4 End of Data. The descriptive procedure call

no data (L)

has the effect of setting the hidden variable H6 to indicate the parameter L. Here L is a label. End of data as defined here has meaning only on input, and it does not refer to any specific hardware features; it occurs when data is requested for input but no more data remains on the corresponding input medium. At this point, a transfer to statement labeled L will occur. If the procedure no data is not used, transfer will occur to a "label" which has effectively been inserted just before the final end in the ALGOL program, thus terminating the program. (In this case the implementor may elect to provide an appropriate error comment.)

B.3.4 End of Data

End of data is defined as the occurrence of an end-of-file condition on the input device.

If the procedure NO DATA was not called, transfer occurs to the label established for the channel by the EOF procedure (Section 3.3). If the EOF procedure was not called, or if the established label is no longer accessible, the object program terminates abnormally with the message UNCHECKED EOF.

B.3.5 Examples. A layout procedure might look as follows:

```
procedure LAYOUT; begin format ('/');
    if B then begin format 1('XB',Y + 10); no data (L32) end;
    h lim (if B then 1 else 10,30) end;
```

Note that layout procedures never have formal parameters; this procedure, for example, refers to three global quantities, B, Y and L32. Suppose Y has the value 3; then this layout accomplishes the following:

Hidden Variable	Procedure	if B = true	if B = false
H1	format	'13B'	'/'
H2	h lim	(1,30)	(10,30)
H3	v lim	(1,∞)	(1,∞)
H4	h end	(, ,)	(, ,)
H5	v end	(, ,)	(, ,)
H6	no data	L32	end program
H7	tabulation	1	1

As a more useful example, we can take the procedure LAYOUT of Section B.1 and rewrite it so that the horizontal margins (11,110) are used on the page, except that if P-overflow or R-overflow occurs we wish to use the margins (16,105) for overflow lines.

```
procedure LAYOUT; begin

    format 1 ('↑,(X(BB-ZZZZ.DD) ,//)' ,n);

    h lim (11,110); h end (K,L,L) end;

procedure K; h lim (11,110);

procedure L; h lim (16,105);
```

This causes the limits (16,105) to be set whenever overflow occurs, and the "/" in the format will reinstate the original margins when it causes procedure K to be called. (If the programmer wishes a more elaborate treatment of the overflow case, depending on the value of P, he may do this using the procedures of Section B.6).

B.4 List Procedures

B.4.1 General characteristics. The concept of a list procedure is quite important to the input-output conventions described in this report, and it may also prove useful in other applications of ALGOL. It represents a specialized application of the standard features of ALGOL which permit a procedure identifier, L, to be given as an actual parameter of a procedure, and which permit procedures to be declared within procedures. The purpose of a list procedure is to describe a sequence of items which is to be transmitted for input or output. A procedure is written in which the name of each item V is written as the argument of a procedure, say ITEM, thus: ITEM(V). When the list procedure is called by an input-output system procedure, another procedure (such as the internal system procedure out item) will be "substituted" for ITEM, V will be called by name, and the value of V will be transmitted for input or output. The standard sequencing of ALGOL statements in the body of the list procedure determines the sequence of items in the list.

A simple form of list procedure might be written as follows:

```
procedure LIST (ITEM);

begin ITEM(A); ITEM(B); ITEM(C) end
```

which says that the values of A, B, and C are to be transmitted.

A more typical list procedure might be:

```
procedure PAIRS (ELT);

for i : = 1 step 1 until n do begin ELT(A[i]);

        ELT(B[i]) end
```

This procedure says that the values of the list of items A[1], B[1], A[2], B[2], . . . , A[n], B[n] are to be transmitted, in that order. Note that if $n \leqslant 0$ no items are transmitted at all.

The parameter of the "item" procedure (i.e., the parameter of ITEM or ELT in the above examples) is called by name. It may be an arithmetic expression, a Boolean expression, or a string, in accordance with the format which will be associated with the item. Any of the ordinary features of ALGOL may be used in a list procedure, so there is great flexibility.

Unlike layout procedures which simply run through their statements and set up hidden variables H1 through H6, a list procedure is executed step by step with the input or output procedure, with control transferring back and forth. This is accomplished by special system procedures such as in item and out item which are "interlaced" with the list procedure, as described in Sections B.4.2 and B.5. The list procedure is called with in item (or out item) as actual parameter, and whenever this procedure is called within the list procedure, the actual input or output is taking place. Through the interlacing, special format control, including the important device-independent overflow procedures, can take place during the transmission process. Note that a list procedure may change the hidden variables by calling a descriptive procedure; this can be a valuable characteristic, e.g. when changing the format, based on the value of the first item which is input.

B.4.2 Other applications. List procedures can actually be used in many ways in ALGOL besides their use with input or output routines; they are useful for manipulating linear lists of items of a quite general nature. To illustrate this fact, and to point out how the interlacing of control between list and driver procedures can be accomplished, here is an example of a procedure which calculates the sum of all of the elements in a list (assuming all elements are of integer or real type):

```
procedure ADD(Y,Z); begin

procedure A(X); Z := Z + X;

Z := 0; Y(A) end
```

The call ADD(PAIRS,SUM) will set the value of SUM to be the sum of all of the items in the list PAIRS defined in Section B.4.1. The reader should study this example carefully to grasp the essential significance of list procedures. It is a simple and instructive exercise to write a procedure which sets all elements of a list to zero.

B.5 Input and Output Calls

Here procedures are described which cause the actual transmission of input or output to take place.

To give a more complete range of input-output procedures, the following calls have been added:

IN CHARACTER	IN REAL	IN ARRAY
OUT CHARACTER	OUT REAL	OUT ARRAY

Character Transmission

The procedures IN CHARACTER and OUT CHARACTER provide the means of communicating between input-output devices and the variables of the program in the terms of basic characters.

```
IN CHARACTER (channel, string, destination)

OUT CHARACTER (channel, string, source)
```

IN CHARACTER examines the next basic character on the channel; if its value corresponds to the external BCD value 00_8, the integer variable destination is set to -1. If not, the character is compared for equality with the characters that comprise the string. If a match is found at the Jth character, destination is set to value J; if no match is found, destination is set to 0.

OUT CHARACTER examines the value of source. If it is negative, the character which corresponds to external BCD 00_8 is output. If the value is in the range of 1 to J, where J is the length of the string, the corresponding character of the string is output; otherwise an object program error results.

In both IN CHARACTER and OUT CHARACTER embedded string quotes ' and ' are each counted as three characters, as in the procedure CHLENGTH (Section 3.2.1).

B.5.2 Transmission of Type real

Transmission of information of type real between variables of the program and an external device may be accomplished by the procedure calls

IN REAL (channel, destination)

OUT REAL (channel, source)

where channel and source are arithmetic expressions and destination is a variable of type real.

The two procedures IN REAL and OUT REAL form a pair. The procedure IN REAL will assign the next value appearing on the input device to the real type variable given as the second parameter. Similarly, procedure OUT REAL will transfer the value of the second actual parameter to the output device.

A value which has been transferred by the call OUT REAL is represented in such a way that the same value, in the sense of numerical analysis (Section 3.3.6), may be transferred back to a variable by means of procedure IN REAL.

The procedures IN REAL and OUT REAL handle numbers in standard format.

B.5.3 Transmission of Arrays

Arrays may be transferred between input-output devices by means of the procedure calls

IN ARRAY (channel, destination)

OUT ARRAY (channel, source)

where channel must be an arithmetic expression and destination and source are arrays of type real.

Procedures IN ARRAY and OUT ARRAY also form a pair; they transfer the ordered set of numbers which form the value of the array, given as the second parameter. The array bounds are defined by the corresponding array declaration rather than by additional parameters (the mechanism for doing this is already available in ALGOL-60 for the value call of arrays).

The order in which the elements of the array are transferred corresponds to the lexicographic order of the values of the subscripts as follows:

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

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

$k_i = j_i$ $(i = 1,2,\ldots,p-1)$

and $k_p < j_p$ $(1 \leq p \leq m)$

It should be recognized that the possibly multidimensional structure of the array is not reflected in the corresponding numbers on the external device where they appear only as a linear sequence as defined above.

The representation of the numbers on the external device conforms to the same rules as given for IN REAL and OUT REAL; in fact it is possible to input numbers by IN REAL which before have been output by OUT ARRAY.

B.5.1 Output

An output process is initiated by the call:

out list (unit,LAYOUT,LIST)

Here unit is an integer parameter called by value, which is the number of an output device (cf. Section B.1.1). The parameter LAYOUT is the name of a layout procedure (Section B.3), and LIST is the name of a list procedure (Section B.4).

There is also another class of procedures, named output n, for n = 0,1,2, . . . , which is used for output as follows:

output n (unit,format string,e_1,e_2, . . . , e_n)

B.5.1 Output

The single procedure with call:

OUTPUT (channel, format string, e_1,e_2, . . . ,e_n)

replaces the n+1 procedures defined with call:

OUTPUT n (channel, format string, e_1,e_2, . . . ,e_n)

n = 0, ∞

The number of e_i variables included in the call to OUTPUT defines to which of the n+1 procedures (defined in the proposal) it is equivalent. For example:

OUTPUT (channel, format string, e_1)

is equivalent to

OUTPUT 1 (channel, format string, e_1)

defined in the proposal.

A call to OUTPUT may include 0-61 variables.

The parameters e_i also may be an array identifier or an implied for-loop (these parameters are explained at the end of Section B.5.1.1, page 3-29).

Each of these latter procedures can be defined in terms of out list as follows:

procedure output n(unit, format string, e_1,e_2, . . . ,e_n);

begin procedure A; format (format string);

procedure B(P); begin P(e_1); P(e_2); . . . ; P(e_n) end;

out list (unit, A,B) end

We will therefore assume in the following rules that out list has been called.

Let the variables ρ and ρ' indicate the current position in the output for the unit under consideration, i.e., lines 1, 2, . . . , ρ' of the current page have been completed, as well as character positions 1,2, . . . ,ρ of the current line (i.e., of line ρ'+1). At the beginning of the program, $\rho = \rho' = 0$. The symbols P and P′ denote the line size and page size (see Section B.2). Output takes place according to the following algorithm:

Step 1. The hidden variables are set to standard values:

H1 is set to the "standard" format ' '.

H2 is set so that L = 1,R = ∞.

H3 is set so that L′ = 1,R′= ∞.

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedure defined as follows: "procedure DUMMY;;".

H5 is set so that $P_{N'}$, $P_{R'}$, $P_{P'}$ are all effectively equal to DUMMY.

H6 is set to terminate the program in case the data ends (this has meaning only on input).

Step 2. The layout procedure is called; this may change some of the variables H1, H2, H3, H4, H5, H6.

Step 3. The next format item of the format string is examined. (Note. After the format string is exhausted, "standard" format, Section A.5, is used from then on until the end of the procedure. In particular, if the format string is ' ', standard format is used throughout.) Now if the next format item is a title format, i.e., requires no data item, we proceed directly to step 4. Otherwise, the list procedure is activated; this is done the first time by calling the list procedure, using as actual parameter a procedure named out item ; this is done on all subsequent times by merely returning from the procedure out item , which will cause the list procedure to be continued from the latest out item call. (Note: The identifier out item has scope local to out list , so a programmer may not call this procedure directly). After the list procedure has been activated in this way, it will either terminate or will call the procedure out item. In the former case, the output process is completed; in the latter case, continue at step 4.

Step 4. Take the next item from the format string. (Notes. If the list procedure was called in step 3, it may have called the descriptive procedure format , thereby changing from the format which was examined during step 3. In such a case, the new format is used here. But at this point the format item is effectively removed from the format string and copied elsewhere so that the format string itself, possibly changed by further calls of format , will not be interrogated until the next occurrence of step 3. If the list procedure has substituted a title format for a nontitle format, the "item" it specifies will not be output, since a title format consists entirely of insertions and alignment marks.)

Set "toggle" to false. (This is used to control the breaking of entries between lines.) The alignment marks, if any, at the left of the format item, now cause process A (below) to be executed for each "/", and process B for each "↑". If the format item consists entirely of alignment marks, then go immediately to step 3. Otherwise the size of the format (i.e. the number of characters specified in the output medium) is determined. Let this size be denoted by S. Continue with step 5.

Step 5. Execute process C, to ensure proper page alignment.

Step 6. Line alignment: If $\rho < L-1$, effectively insert blank spaces so that $\rho = L - 1$. Now if toggle = true, go to step 9; otherwise, test for line overflow as follows: If $\rho + S > R$, perform process D, then call P_R and go to step 8; otherwise, if $\rho + S > P$, perform process D, call P_P, and go to step 8.

Step 7. Evaluate the next output item and output it according to the rules given in Section A; in the case of a title format, this is simply a transmission of the insertions without the evaluation of an output item. The pointer ρ is set to $\rho + S$. Any alignment marks at the right of the format item now cause activation of process A for each "/" and of process B for each "↑". Return to step 3.

Step 8. Set toggle to true. Prepare a formatted output item as in step 7, but do not record it on the output medium yet (this is done in step 9). Go to step 5. (It is necessary to re-examine page and line alignment, which may have been altered by the overflow procedure; hence we go to step 5 rather than proceeding immediately to step 9.)

Step 9. Transfer as many characters of the current output item as possible into positions $\rho + 1, \ldots$, without exceeding position P or R. Adjust ρ appropriately. If the output of this item is still unfinished, execute process D again, call P_R (if $R \leq P$) or P_P (if $P < R$), and return to step 5. The entire item will eventually be output, and then we process alignment characters as in step 7, finally returning to step 3.

Process A. ("/" operation) Check page alignment with process C, then execute process D and call procedure P_N.

Process B. ("↑" operation) If $\rho > 0$, execute process A. Then execute process E and call procedure $P_{N'}$.

Process C. (Page alignment)

If $\rho' < L' - 1$ and $\rho > 0$: execute process D, call procedure P_N, and repeat process C.
If $\rho' < L' - 1$ and $\rho = 0$: execute process D until $\rho' = L' - 1$.
If $\rho' + 1 > R'$: execute process E, call procedure $P_{R'}$, and repeat process C.
If $\rho' + 1 > P'$: execute process E, call procedure $P_{P'}$, and repeat process C.

Process D. Skip the output medium to the next line, set $\rho = 0$, and set $\rho' = \rho' + 1$.

Process E. Skip the output medium to the next page, and set $\rho' = 0$.

Steps 1-9 and Process A-E have been implemented as follows:

Step 1. (Initialization)

The hidden variables are set to standard values:

H1 is set to the standard format ' '.

H2 is set so that L = 1, R = ∞.

H3 is set so that L′ = 1, R′ = ∞.

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedure defined as follows: "procedure DUMMY;;".

H5 is set so that $P_{N'}$, $P_{R'}$, $P_{P'}$ are all effectively equal to DUMMY.

H7 is set so that TAB = 1.

Step 2. (Layout)

The layout procedure is called; this may change some of the variables H1,H2,H3,H4,H5,H7. Set T to false. (T is a Boolean variable used to control the sequencing of data with respect to title formats; T = true means a value has been transmitted to the procedure which has not yet been output.)

Step 3. (Communication with list procedure)

The next format item of the format string is examined. (Note. After the format string is exhausted, standard format, Section A.5, is used until the end of the procedure. In particular, if the format string is ' ', standard format is used throughout.) If the next format item is a title format (requires no data item), proceed directly to step 4. If T = true proceed to step 4. Otherwise, the list procedure is activated; this is initiated by calling the list procedure, using a procedure named OUT ITEM as the actual parameter. Each subsequent return from the procedure OUT ITEM will cause the list procedure to be continued from the latest OUT ITEM call. Since the scope of the identifier OUT ITEM is local to OUT LIST, this procedure cannot be called directly.

After the list procedure has been activated it will either terminate or call the procedure OUT ITEM. If it terminates the output process is completed. If the procedure OUT ITEM is called, T is set to true and any assignments to hidden variables that may have been made by calls on list procedures will cause adjustment to the variables H1,H2,H3,H4,H5,H7, which are local to OUT ITEM, the procedure will then continue at step 4.

Step 4. (Alignment marks)

If the next format item includes alignment marks at its left, process A (a subroutine below) is executed for each /, process B for each ↑ and process C for each J.

Step 5. (Get within margins)

Process G is executed to ensure proper page and line alignment.

Step 6. (Formatting the output)

The next item is taken from the format string.

In unusual cases, the list procedure or an overflow procedure may have called the descriptive procedure format, thereby changing the format string. If so, the new format string is examined from the beginning; and it is conceivable that the format items examined in steps 3, 4, 6 might be three different formats. At this point, the current format item is effectively removed from the format string and copied elsewhere; so that the format string itself, possibly changed by further calls of format, will not be interrogated until the next occurrence of step 3.

Alignment marks at the left of the format item are ignored. If the format item is not composed only of alignment marks and insertions, the value of T is examined. If T = false, action is undefined (a nontitle format has been substituted for a title format in an overflow procedure, and this is not allowed). Otherwise, the output item is evaluated and T is set to false. The rules of format are applied and the characters $x_1x_2...x_s$, which represent the formatted output on the external device, are determined.

Step 7. (Check for overflow)

If $\rho + s \leq R$ and $\rho + s \leq P$, where s is the size of the item as determined in step 6, the item will fit on this line, so step 9 is executed.

Step 8. (Processing of overflow)

Process H (ρ + s) is performed. Then if $\rho + s \leq R$ and $\rho + s \leq P$, step 9 is executed; otherwise K is set to min (R, P) - ρ. $x_1x_2 \ldots x_k$ is output, ρ is set = min (R, P), and $x_1x_2 \ldots x_{s-k}$ to $x_{k+1}\ x_{k+2} \ldots x_s$. s is decreased by k and step 8 repeated.

Step 9. (Finish the item)

$x_1x_2 \ldots x_s$ is output, and ρ increased by s. Any alignment marks at the right of the format item now cause activation of process A for each /, process B for each ↑, and process C for each J. Return to step 3.

Process A (/ operation)

Page alignment is checked with process F, and process D is executed. Procedure P_N is called.

Process B (↑ operation)

If $\rho > 0$, process A is executed. Process E is then executed and procedure $P_{N'}$ called.

Process C (J operation)

Page and line alignment are checked with process G. Then K is set = $((\rho - L + 1) \div TAB + 1) \times TAB + L - 1$ (the next tab setting for ρ), where TAB is the tab spacing for this channel. If $k \geq$ min (R, P), process H(k) is performed; otherwise blanks are inserted until $\rho = k$.

Process D (New line)

The output device is skipped to the next line, ρ is set to 0, and ρ' is set to $\rho' + 1$.

Process E (New page)

The output device is skipped to the next page, and ρ is set to 0. Skipping the output device to a new page involves setting character 1 of the next line to a print control character. On a normal line, character 1 is set to a value which results in single spacing. This character does not appear if the external device is a printer; on any other device, it is the first character on the external device. When paging is specified, character 1 is not available for use, regardless of the external device. To overcome the loss of this character position, the procedure H LIM (Section B.3.2) increases the values of the L and R parameters by 1.

If no paging is specified, the user may reference character 1; H LIM does not adjust the L and R parameter values. However, if the external device is a printer, character 1 of each record is interpreted by the driver to control page ejection and to control random page and line skipping. The user should set this character to avoid loss of a significant character.

Process F (Page alignment)

If $\rho' + 1 < L'$ process D is executed until $\rho' = L' - 1$. If $\rho' + 1 > R'$: process E is executed, $P_{R'}$ is called, and process F is repeated. If $\rho' + 1 > P'$: process E is executed, $P_{P'}$ is called, and process F repeated. This process must terminate because $1 \leq L' \leq R'$ and $1 \leq L' \leq P'$. (If a value $L' > P$ is chosen, L' is set equal to 1.)

Process G (Page and line alignment)

Process F is executed. Then, if $\rho + 1 < L$: blank spaces are output until $\rho + 1 = L$. If $\rho + 1 > R$ or $\rho + 1 > P$: process H ($\rho + 1$) is performed. This process must terminate because $1 \leq L \leq R$ and $1 \leq L \leq P$. If a value of $L > P$ is chosen, L is set equal to 1.

Process H(k) (Line overflow)

Process D is performed. If $k > R$, P_R is called; otherwise P_P is called. Then process G is performed to ensure page and line alignment. Note: upon return from any of the overflow procedures, any assignments to hidden variables made by calls on descriptive procedures will cause adjustment to the corresponding variables H1,H2,H3,H4,H5,H7 local to OUT ITEM.

B.5.1.1 OUTPUT Parameters

The parameter list of variables e_i may be of two additional kinds: array identifiers or implied for-loops.

Array Identifier

An OUTPUT procedure with an array identifier parameter is the same as an OUT ARRAY procedure (Section B.5.3) except that formatting is performed according to the relevant items of the format string.

Implied For-Loop

Some terms in the following syntax are new and do not appear on previous pages of this manual in the ALGOL Report.

Syntax

<for control clause> ::= <loop control> := <initial> : <final> |

<loop control> := <initial> : <final> : <step>

<loop control> ::= <identifier>

<initial> :: = <arithmetic expression>

<final> :: = <arithmetic expression>

<step> :: = <arithmetic expression>

<basic loop> :: = (<actual parameter list>, <for control clause>)

<loop element> ::= <actual parameter> | <basic loop>

<loop list> ::= <loop element> | <loop list>, <loop element>

<implied for-loop> ::= <basic loop> |

(<loop list>, <for control clause>)

Semantics

The implied for-loop can be considered as a condensed form of the body of the procedure LIST used in the procedure OUTLIST. The body, in this instance, being a for statement with step-until-elements, which is interpreted according to the rules of section 4.6.4.2 of the report. This statement may embody other for statements to a maximum depth of 6. The initial, final, and step values are evaluated once only on entry to the loop, and thereafter remain unchanged. When the step is absent, its implied value is 1. The elements of the loop list are generally (but not necessarily) functions of the control variable which must be type real or integer.

In the examples below use the function item, which is the name used for the parameter of LIST in section B.4 describing list procedures.

Some implied for-loops might be:

```
(j, B[j], j:=1:N)
```

which implies:

```
for j:=1 step 1 until N do
begin item (j); item (B[j]) end;
```

```
((A [i,j], i:=1:M), j:=1:N)
```

which implies:

```
for j:=1 step 1 until N do
for i:=1 step 1 until M do
    item (A[i,j]);
```

```
(theta, sin (theta), theta:=0:pi/2:10↑-2)
```

which implies:

```
for theta:=0 step 10↑-2 until pi/2 do
begin item (theta) ; item (sin (theta)) end;
```

B.5.2 Input

The input process is initiated by the call:

in list (unit,LAYOUT,LIST)

The parameters have the same significance as they did in the case of output, except that unit is in this case the number of an input device. There is a class of procedures input n which stand for a call with a particularly simple type of layout and list, just as discussed in Section B.5.1 for the case of output. In the case of input, the parameters of the "item" procedure within the list must be variables.

B.5.2 Input

The single procedure with call:

INPUT (channel, format string, $e_1, e_2, \ldots e_n$)

replaces the n+1 procedures defined in the proposal with call:

INPUT n (channel, format string, $e_1, e_2, \ldots e_n$)

$n = 0, \infty$

The number of e_i variables included in the call to INPUT defines to which of the n+1 procedures (defined in the report) it is equivalent. For example:

INPUT (channel, format string, e_1, e_2, e_3)

is equivalent to

INPUT 3 (channel, format string, e_1, e_2, e_3)

defined in the proposal.

A call to INPUT may include 0-61 variables.

The parameter e_i also may be an array identifier or an implied for-loop. INPUT procedures with array identifier or implied for-loop parameters perform in the same way as OUTPUT (Section B.5.1.1).

The various steps which take place during the execution of in list are very much the same as those in the case of out list, with obvious changes. Instead of transferring characters of title format, the characters are ignored on input. If the data is improper, some standard error procedure is used. (cf. Section A.1.3.8).

The only significant change occurs in the case of standard input format, in which the number S of the above algorithm cannot be determined in step 4. The tests $\rho + S > R$ and $\rho + S > P$ now become a test on whether positions $\rho + 1, \rho + 2, \ldots,$ min(R,P) have any numbers in them or not. If so, the first number, up to its delimiter, is used; the R and P positions serve as delimiters here. If not, however, overflow occurs, and subsequent lines are searched until a number is found (possibly causing additional overflows). The right boundary min(R,P) will not count as a delimiter in the case of overflow. This rule has been made so that the process of input is dual to that of output: an input item is not split across more than one line unless it has overflowed twice. Notice that the programmer has the ability to determine the presence or absence of data on a card when using standard format, because of the way overflow is defined. The following program, for example, will count the number n of data items on a single input card and will read them into A[1] ,A [2] , . . . , A[n]. (Assume unit 5 is a card reader.)

```
procedure LAY; h end (EXIT,EXIT,EXIT);

procedure LIST(ITEM); ITEM(A[n + 1] );

procedure EXIT; go to L2;

n : = 0; L1: in list(5,LAY,LIST); n : = n + 1; go to L1;

L2:; comment mission accomplished;
```

Steps 1-9 and A - E of input have been implemented as follows:

Step 1. (Initialization)

The hidden variables are set to standard values:

H1 is set to the standard format ' '.

H2 is set so that L = 1, R = ∞.

H3 is set so that L′ = 1, R′ = ∞.

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedure defined as follows: "procedure DUMMY;;".

H5 is set so that $P_{N'}$, $P_{R'}$, $P_{P'}$ are all effectively equal to DUMMY.

H6 is set to terminate the program in case the data ends.

H7 is set so that TAB = 1.

Step 2. (Layout)

The layout procedure is called; this may change some of the variables H1,H2,H3,H4,H5,H6,H7. Set T to false. T is a Boolean variable used to control the sequencing of data with respect to title formats; T = true means a value has been requested of the procedure which has not yet been input.

Step 3. (Communication with list procedure)

The next format item of the format string is examined. (After the format string is exhausted, standard format is used until the end of the procedure. In particular, if the format string is ' ', standard format is used throughout.) If the next format item is a title format, (requires no data item) step 4 is executed directly. If T = true step 4 is executed. Otherwise, the list procedure is activated; this is initiated by calling the list procedure, using a procedure named IN ITEM as the actual parameter. Each subsequent return from the procedure IN ITEM, will cause the list procedure to be continued from the latest IN ITEM call. Since the scope of the identifier IN ITEM is local to IN LIST, this procedure cannot be called directly. After the list procedure has been activated, it will either terminate or it will call the procedure IN ITEM. In the former case, the input process is completed; in the latter case, T is set to true and any assignments to hidden variables resulting from the list procedure will cause adjustments to the variables H1, H2, H3, H4, H5, H6, H7 (which are local to IN ITEM) and will then continue at step 4.

Step 4. (Alignment marks)

If the next format item includes alignment marks at its left, process A is executed (a subroutine below) for each / , process B for each ↑ , and process C for each J.

Step 5. (Get within margins)

Process G is executed to ensure proper page and line alignment.

Step 6. (Formatting for input)

The next item is taken from the format string. In unusual cases, the list procedure or an overflow procedure may have called the descriptive procedure format, thereby changing the format string. In such cases, the new format string is examined from the beginning; it is conceivable that the format items examined in steps 3, 4, 6 might be three different formats. At this point, the current format item is effectively removed from the format string and copied elsewhere; so that the format string itself, possibly changed by further calls of format, will not be interrogated until the next occurrence of step 3.

Alignment marks at the left of the format item are ignored. If the format item is not composed only of alignment marks and insertions, the value of T is examined. If T = false, undefined action takes place (a nontitle format has been substituted for a title format in an overflow procedure, and this is not allowed). Otherwise, T is set to false.

Step 7. (Check for overflow)

If the present item uses N format, the character positions $\rho + 1$, $\rho + 2$, ... are examined until either a delimited number has been found, (in which case ρ is advanced to the position following the number, and step 9 is executed) or position min (R, P) has been reached with no sign, digit, decimal point, or $_{10}$ encountered. In this case, step 8 is executed with ρ = min (R, P). If N format is not used, step 8 is executed if $\rho + s >$ min (R, P), or step 9 if $\rho + s \leq$ min (R, P).

Step 8. (Processing of overflow)

Process H ($\rho + s$) is performed and the following procedure:

N format: Characters are input until a number followed by a delimiter is found and step 9 is executed; or if position min (R, P) is reached, a partial number may have been examined. Step 8 is repeated until a number followed by a delimiter has been input.

Other: If $\rho + s \leq R$ and $\rho + s \leq P$, step 9 is executed; otherwise input k = min (R, P) $-\rho$ characters, set ρ = min (R, P) decrease s by k, and repeat this step.

Step 9. (Finish the item)

If any format other than N is being used, s characters are input. The value of the item that was input here is determined (steps 7 and 8 in the case of N format) using the rules of format. This value is assigned to the actual parameter of IN ITEM unless a title format was specified. ρ is increased by s. Any alignment marks at the right of the format item now cause activation of process A for each / , process B for each ↑, and process C for each J. Return to step 3.

Process A (/ operation)

Page alignment is checked with process F, process D is executed and procedure P_N called.

Process B (↑ operation)

If $\rho \geq 0$, process A is executed. Then process E is executed and procedure $P_{N'}$ called.

Process C. (J operation)

Page and line alignment are checked with process G. Then let $k = ((\rho - L + 1) \div TAB + 1) \times TAB + L - 1$ (the next tab setting for ρ), where TAB is the tab spacing for this channel. If $k \geq$ min (R, P), process H(k) is performed; otherwise character positions are skipped until $\rho = k$.

Process D. (New line)

The input medium is skipped to the next line, ρ is set to 0, and ρ' is set to $\rho' + 1$.

Process E. (New page)

The input medium is skipped to the next page, and ρ' is set to 0. Skipping the input device to a new page involves the assumption that the next line on the input device begins the new page (control character in position 1 which is not accessible by the program) as specified in the corresponding process of output.

Process F. (Page alignment)

If $\rho' + 1 < L'$ process D is executed until $\rho' = L' - 1$. If $\rho' + 1 > R'$: process E is executed, $P_{R'}$ called, and process C repeated. If $\rho' + 1 > P$: process E is executed, $P_{P'}$ called, and process C repeated. This process must terminate because $1 \leq L' \leq R'$ and $1 \leq L' \leq P$. (If a value of $L' > P$ is chosen, L' is set equal to 1.)

Process G. (Page and line alignment)

Process F is executed. Then, if $\rho + 1 < L$: character positions are skipped until $\rho + 1 = L$. If $\rho + 1 > R$ or $\rho + 1 > P$: process H ($\rho + 1$) is performed. This process must terminate because $1 \leq L \leq R$ and $1 \leq L \leq P$. If a value of $L > P$ is chosen, L is set equal to 1.

Process H(k) (Line overflow)

Process D is performed. If k>R, P_R is called; otherwise Pp is called. Then process G is performed to ensure page and line alignment. NOTE: Upon return from any of the overflow procedures, any assignments to hidden variables that have been made by calls on descriptive procedures, will cause adjustments to the corresponding variables, H1, H2, H3, H4, H5, H6, H7 local to IN ITEM.

B.5.3 Skipping

Two procedures are available which achieve an effect similar to that of the "tab" key on a typewriter:

h skip (position, OVERFLOW)

v skip (position, OVERFLOW)

where position is an integer variable called by value, and OVERFLOW is the name of a procedure. These procedures are defined only if they are called within a list procedure during an in list or out list operation. For h skip, if $\rho <$ position, set ρ = position; but if $\rho \geq$ position, call the procedure OVERFLOW. For v skip, an analogous procedure is carried out: if $\rho' <$ position, effectively execute process A of Section B.5.1 (position $- \rho'$) times; but if $\rho' \geq$ position, call the procedure OVERFLOW.

B.5.3 Skipping

The procedures H SKIP and V SKIP have been replaced by the procedure TABULATION, described in Section B.3.3.

B.5.4 Intermediate data storage

The procedure call

put (n, LIST)

where n is an integer parameter called by value and LIST is the name of a list procedure (Section B.4), takes the value specified by the list procedure and stores them, together with the identification number n. Anything previously stored with the same identification number is lost. The variables entering into the list do not lose their values

The procedure call

get (n, LIST)

where n is an integer parameter called by value and LIST is the name of a list procedure, is used to retrieve the set of values which has previously been put away with identification number n. The items in LIST must be variables. The stored values are retrieved in the same order as they were placed, and they must be compatible with the type of the elements specified by LIST; transfer functions may be invoked to convert from real to integer type or vice versa. If fewer items are in LIST than are associated with n, only the first are retrieved; if LIST contains more items, the situation is undefined. The values associated with n in the external storage are not changed by get.

B.5.4 Intermediate Data Storage

The procedures GET and PUT are not implemented; they have been replaced by GET ARRAY and PUT ARRAY, although these are in no way analogous. The calls are:

GET ARRAY (channel,destination)

PUT ARRAY (channel,source)

Destination and source are the names of arrays. These procedures can be used only on channels defined with the character A on the channel card (Chapter 7).

GET ARRAY reads one record of the same length as destination directly from the channel into destination. The record is not stored first in a format area, and no regard is made for maximum record size or paging. The record should contain the array arranged by rows (as defined in Transmission of Array).

PUT ARRAY writes one record, equal in length to source, directly from source to the channel. The record is not stored first in a format area and no regard is made for maximum record size or paging. The record reflects exactly how the array is stored in memory, by rows.

B.6 Control Procedures

The procedure calls

out control (unit, x1,x2,x3,x4)

in control (unit, x1,x2,x3,x4)

may be used by the programmer to determine the values of normally "hidden" system parameters, in order to have finer control over input and output. Here unit is the number of an output or input device, and x1,x2,x3,x4 are variables. The action of these procedures is to set x1,x2,x3,x4 equal to the current values of ρ,P, ρ',P', respectively, corresponding to the device specified.

B.6 Control Procedure

In the input-output system as described up to this point, the physical limits characteristic of the various devices (P, P′), the number of spaces (k) which serves as a number delimiter in standard format, and the current value of the character pointers (ρ, ρ'), are effectively system parameters which are not directly accessible to the programmer. These quantities are accessible to, and in many cases modified by, the several input and output procedures.

To obtain finer control over the input-output processes, the programmer can gain access to these quantities through the procedure call

SYSPARAM (channel,function,quantity)

Channel is an arithmetic expression called by value specifying the input-output device concerned.

Function is an arithmetic expression called by value specifying the particular quantity to be accessed, and specifying whether that quantity is to be interrogated or changed.

Quantity is an integer variable called by name which will either represent the new value or be assigned the present value of the quantity dependent on function.

The following list defines the standard set of quantities accessible through SYSPARAM and the corresponding value of function.

For the external device associated with channel:

If function = 1, quantity : = ρ;

If function = 2, ρ : = quantity†

If function = 3, quantity : = ρ';

If function = 4, ρ' : = quantity†

If function = 5, quantity : = P;

If function = 6, P : = quantity††

If function = 7, quantity : = P';

If function = 8, P' : = quantity††

If function = 9, quantity : = k;

If function = 10, k : = quantity

ρ and ρ' are the character and line pointers

P and P' are the physical limits of the device

k is the number of blanks delimiting a standard number format

B.7 Other Procedures

Other procedures which apply to specific input or output devices may be defined at installations, (tape skip and rewind for controlling magnetic tape, etc.). An installation may also define further descriptive procedures (thus introducing further hidden variables); for example, a procedure might be added to name a label to go to in case of an input error. Procedures for obtaining the current values of hidden variables might also be incorporated.

B.7 Other Procedures

The following additional procedures have been implemented. They are described fully in 3.3 and 3.4.

†Since ρ and ρ' represent the actual (physical) positions on the external device, function = 2 or 4 will generally cause some action to take place for that device. When setting ρ if quantity > ρ, insert blanks until ρ = quantity. If quantity $\leq \rho$ perform a line advance operation, set ρ = 0 and insert blanks until ρ = quantity. When setting ρ' if quantity > ρ' perform line advance operations until ρ' = quantity. If quantity $\leq \rho'$ skip to next page, set ρ' = 0, and perform line advance operations until ρ' = quantity.

††These operations change the physical limits for the input-output device where this is possible (e.g., block length on magnetic tape). When these limits cannot be changed for the input-output device these functions are equivalent to a dummy statement.

Primitive Procedures	CHLENGTH STRING ELEMENT
Control Procedures	ARTHOFLW (label) PARITY (channel, label) EOF (channel, label) BAD DATA (channel, label)
Hardware Function Procedures	SKIPF (channel) SKIPB (channel) ENDFILE (channel) REWIND (channel) UNLOAD (channel) BACKSPACE (channel)
Miscellaneous Procedures	IOLTH (channel) MODE (channel, type) CONNECT (channel, array, label) DUMP (identifying integer) CLOCK
ECS procedures †	READ ECS (address, array, label) WRITE ECS (address, array, label)
Interactive procedures †	CONNEC (channel) DISCON (channel)

C. An Example

A simple example follows, which is to print the first 20 lines of Pascal's triangle in triangular form:

```
          1
        1   1
      1   2   1
    1   3   3   1
```

These first 20 lines involve numbers which are at most five digits in magnitude. The output is to begin a new page, and it is to be double-spaced and preceded by the title "PASCALS TRIANGLE". We assume that unit number 3 is a line printer.

Two solutions of the problem are given, each of which uses slightly different portions of the input-output conventions.

†Not available under KRONOS.

```
begin integer N, K, printer;

        integer array A[0:19];
        procedure AK (ITEM); ITEM (A[K]);
        procedure TRIANGLE; begin format ('6Z'); h lim (58 - 3 × N, 63 + 3 × N)
        comment for values of N less than 3 the h lim call will be ignored in this implementation. (See B.3.2);
        end;


printer := 3;
output 0 (printer, '↑'PASCALS␣TRIANGLE'//');
for N := 0 step 1 until 19 do
        begin A[N] := 1;
        for K := N - 1 step -1 until 1 do A[K] := A[K - 1] + A[K];
        for K := 0 step 1 until N do out list (printer,TRIANGLE,AK);
        output 0 (printer, '//') end end

begin integer N,K, printer;
        integer array A[0:19];
        procedure LINES;format 2('XB,X(6Z),//',57-3×N,N+1);
        procedure LIST(Q); for K := 0 step 1 until N do Q(A[K]);
printer := 3;
output 1 (printer, '↑20S//','PASCALS␣TRIANGLE');
for N := 0 step 1 until 19 do
        begin A [N] := 1;
        for K := N - 1 step -1 until 1 do A[K] := A[K - 1] + A[K];
        out list (printer,LINES,LIST) end end
```

D. Machine-dependent Portions

Since input-output processes must be machine-dependent to a certain extent, the portions of this proposal which are machine-dependent are summarized here.

1. The values of P and P' for the input and output devices.
2. The treatment of I,L, and R (unformatted) format.
3. The number of characters in standard output format.
4. The internal representation of alpha format.
5. The number of spaces, K, which will serve to delimit standard input format values.

REFERENCES

Naur, P.(Ed.) Revised report on the algorithmic language ALGOL-60 Comm. ACM 6 (1963), 1-17.

Extended ALGOL reference manual for the Burroughs B-5000. No. 5000-2102, Burroughs Corp., Detroit, 1963.

SHARE ALGOL-60 translator manual. No. 1426,1577,SHARE Distr. Agency. Oak Ridge ALGOL compiler for the Control Data 1604 computer. Oak Ridge Nat. Lab., Oak Ridge, Tenn.

Duncan, F. G. Input and output for ALGOL-60 on KDF 9. Comp. J. 5 (1963), 341-344.

Hoare, C. A. R. The Elliott ALGOL input/output system. Comp. J. 5 (1963), 345-348.

McCracken, D. D. Guide to ALGOL programming. Wiley, New York, 1962. AED compiler. Electronic Systems lab., MIT, Cambridge, Mass.

Ingerman, P. Z. A syntax-oriented compiler, etc. U of Penn., Moore School of Elect. Engineering, Philadelphia, Pa. 1963.

Ingerman, P. Z., and Merner, J. N. Revised revised ALGOL-60 report. Unpublished.

Perlis, A. J. A format language. Comm. ACM 7 (1964), 89-97.

Baumann, R. ALGOL-Manual der ALCOR-Gruppe. Elektron, Rechen. H. 5/6 (1961), H.2 (1962).

3.2 ADDITIONAL INPUT-OUTPUT PROCEDURES

An additional set of primitive procedures exists without declaration, as follows:

```
CHLENGTH (string)
STRING ELEMENT (s1,i,s2,x)
```

3.2.1 CHLENGTH

CHLENGTH is an integer procedure with a string as a parameter. The value of CHLENGTH (string) is equal to the number of characters of the open string enclosed between the outermost string quotes. It is introduced to make it possible to calculate the length of a given (actual or formal) string. Each embedded string quote counts as three characters, because the 48-character representation of the ALGOL symbol ‘ is '(' and ’ is ')' (see Table 1, Chapter 4).

3.2.2 STRING ELEMENT

The procedure STRING ELEMENT is introduced to enable the scanning or interpretation of a given string (actual or formal) in a machine independent manner. It assigns to the integer variable x an integer corresponding to the ith character of the string s1 as encoded by the string s2.

Effectively an OUT CHARACTER (Section B.5) process is performed on the string s1 according to the integer variable i. An IN CHARACTER process is then performed with the resultant character on the string s2, producing an integer value to be stored in the integer variable x.

3.3 CONTROL PROCEDURES

†ARTHOFLW (label)
PARITY (channel,label)
EOF (channel,label)
BAD DATA (channel,label)

Each one of these procedures establishes a label to which control transfers in the event of an arithmetic error (overflow, underflow or division fault), irrecoverable parity error, end-of-file condition, or mismatch of input data and the corresponding format. Each procedure can be called as many times as necessary to modify the label in the course of a program. PARITY, EOF, and BAD DATA must be called once for each channel for which a label is to be established. If a procedure has not been called; or if the label is no longer accessible when the corresponding condition occurs, the object program terminates abnormally with an error message.

If IN LIST is in operation, a label may be established by the NO DATA procedure (Section B.3.4) instead of by the EOF procedure. During the execution of the IN LIST procedure, any label established by NO DATA procedure takes precedence over an EOF label.

†An ARTHOFLW label can be established but control will not transfer to this label.

3.4 HARDWARE FUNCTION PROCEDURES

A channel is input if last used for a read operation, output if last used for a write operation, and closed if not previously referenced of if referenced by a closing procedure such as ENDFILE.

If any of the following procedures are called for an external device which cannot perform the operation, the procedure is treated as a dummy procedure; and at the completion of the procedure, the channel is considered to be closed. On mass-storage devices the procedures REWIND and UNLOAD position the external device to the beginning of the information.

SKIPF (channel)

This procedure spaces the external device forward past one end-of-file mark. It is treated as a dummy procedure on an output channel. If the channel is associated with a mass-storage device, the procedure is treated as a dummy procedure.

SKIPB (channel)

This procedure spaces the external device backwards past one end-of-file mark. On an output channel before the spacing occurs, any information in the format area is written out and an end-of-file mark is written and backspaced over. If the channel is associated with a mass-storage device, the procedure is treated as a dummy procedure.

ENDFILE (channel)

This procedure writes an end-of-file mark on the external device. It is treated as a dummy procedure on an input channel. Before the end-of-file mark is written, any information in the format area is written out.

REWIND (channel)

This procedure rewinds the external device to load point. On output before rewind occurs, any information in the format area is written out; and an end-of-file mark is written and backspaced over.

UNLOAD (channel)

This procedure unloads the external device. On output before unloading occurs, any information in the format area is written out; and an end-of-file mark is written and backspaced over.

BACKSPACE (channel)

The procedure backspaces past one line on a non-A type channel and one SCOPE logical record on an A type channel. On output before the backspace occurs, any information in the format area is written out; and an end-of-file mark is written and backspaced over.

3.5 MISCELLANEOUS PROCEDURES

IOLTH (channel)

This procedure can be used only on non-formatted channels (those used for GET ARRAY and PUT ARRAY). It yields the number of array elements in the last read or write operation on the external device (the number in the last GET ARRAY or PUT ARRAY operation).

MODE (channel, type).

This procedure sets density or parity for the subsequent reading or writing of the external device. Density and parity are initialized on a channel card and depend on the value of TYPE, as follows:

0 No density or parity selection required

1 Do not change density, set parity to odd (binary)

2 Do not change density, set parity to even (BCD)

3 No density selection required, do not change parity

4 Set density to low (200 bpi), do not change parity

5 Set density to medium (556 bpi), do not change parity

6 Set density to high (800 bpi), do not change parity.

CONNECT (channel, array, label)

This procedure is used to connect the array ARRAY to the channel specified by CHANNEL, so that the array may be used as the formatting area of the channel (Chapter 7). If, for any reason, the connection cannot be made (e. g., the array size is too small to encompass the desired formatting area) an exit to the label LABEL is taken.

Upon exiting the block in which the array ARRAY is declared, the channel CHANNEL is returned to its closed state. On output, any information in the formatting area is written out and an end-of-file mark is written and backspaced over.

DUMP (identifying integer)

This procedure may be used to obtain output of the local (and formal) variables in the currently active block (procedure body). The format is that of the object-time abnormal termination dump (Chapter 12). The dump is entitled:

THIS IS DUMP NUMBER <identifying integer> AT LINE <line number>

Identifying integer is an integer type variable, and the number is modulo 8192. The line number is the source line number from which DUMP was called.

CLOCK

This is a real procedure whose value at any instant is the elapsed CPU time in seconds at the control point at which the ALGOL program is executing. The value is accurate to one millisecond.

3.6 INPUT-OUTPUT ERRORS

At object-time, two types of errors not directly concerned with programming are detected: illegal input-output operation requests and invalid transmission of data (Chapter 8).

3.6.1 ILLEGAL INPUT-OUTPUT OPERATIONS

The object program terminates abnormally with a diagnostic if:

An input (output) operation is requested on a channel associated with a device which cannot read (write), or on a device which is prevented by the operating system from reading (writing).

If the last operation on the channel was neither an input (output) nor a closing operation (such as REWIND for input).

3.6.2 TRANSMISSION ERRORS

Transmission errors are first treated by standard recovery procedures. If an error persists, it is irrecoverable.

On an irrecoverable parity error, control transfers to the label established for the channel by the PARITY procedure. If the PARITY procedure was not called or if the established label is no longer accessible, the object program terminates abnormally with the diagnostic UNCHECKED PARITY.

3.7 END-OF-FILE

When an end-of-file is encountered on an external input device, control transfers to the label established for the channel by the NO DATA procedure (within IN LIST only) or the EOF procedure. If neither procedure has been called and if a label established by either is no longer accessible, the object program terminates abnormally with the message UNCHECKED EOF. During execution of the IN LIST procedure, any label established by NO DATA takes precedence over a label established by EOF.

3.8 END-OF-TAPE

If an end-of-tape is detected during writing, the standard system end of tape procedure is executed.

3.9 ECS PROCEDURES †

WRITE ECS (ADDRESS, ARRAY, LABEL)

This procedure transfers the contents of array, ARRAY, to the user direct access area of ECS beginning at the first word address given by the value of the integer expression, ADDRESS. Control transfers to the label, LABEL, if an irrecoverable parity error occurs. If the parameter, LABEL, is omitted, irrecoverable parity error will cause an abnormal termination dump.

READ ECS (ADDRESS, ARRAY, LABEL)

This procedure transfers the contents of successive locations from the user direct access area of ECS addressed by the integer expression, ADDRESS, into the array, ARRAY, in central memory. Control transfers to the label, LABEL, if an irrecoverable parity error occurs. If the parameter, LABEL, is omitted, irrecoverable parity error will cause an abnormal termination dump.

3.10 INTERACTIVE PROCEDURES †

When an ALGOL-60 program is submitted under INTERCOM, the program and the terminal may communicate through the following input-output procedures:

INPUT, OUTPUT, INREAL, OUTREAL, INARRAY, OUTARRAY, INCHARACTER, OUTCHARACTER, INLIST and OUTLIST.

If the RUN command (INTERCOM) was used to initiate ALGOL execution, channels 60 and 61 are connected automatically to the terminal and any of the above procedures may be used. If, however, input is required on a file which has not been connected to the terminal, either by the CONNECT command (INTERCOM) or by the RUN command as for INPUT and OUTPUT, the following procedures must be used to initiate and terminate program-terminal communications.

CONNEC (CHANNEL)

The channel indicated by the value of the integer expression, CHANNEL, is connected to the terminal initiating the current program. Channel definition must conform to the following rules:

The device type of the file must be allocatable.

The channel must be R type (records on the channel must be delimited by a low order zero-byte record mark).

For output to the terminal, the channel should be PP type (paged).

For input from the terminal, the channel must be non-paged or specified as PP0.

The channel may not be A type (it must have a formatting area).

†Not available under KRONOS.

If the program is not running at an INTERCOM control point, CONNEC has no effect.

DISCON (CHANNEL)

The channel indicated by the value of the integer expression, CHANNEL, is disconnected from the terminal initiating the current program. No further program-terminal interaction may take place on this channel which will revert to an allocatable device type. If the program is not running at an INTERCOM control point, DISCON has no effect.

Notes: A user logical record (line) may be the subject of several input-output functions within an ALGOL program, and since terminal transactions are performed on a line-by-line basis the effects will be as follows:

An output procedure will cause terminal output only upon a line action either by overflow of the specified terminal line length or by means of an explicit format item.

The program will not wait for the terminal user when an input procedure is executed unless the last terminal input exhausted all the data entered from the terminal on that occasion.

Before control is transferred to an ALGOL program for which channels 60 and 61 are connected, the terminal user must enter channel definitions (CHANNEL cards) or execution time options (OPTIONS cards) from the terminal. Execution will not begin until the user signifies the end of these items.

INPUT TO COMPILATION 4

Input to the compiler may be an ALGOL source program or an ALGOL source procedure. More than one source program or source procedure may be compiled with a single call of the compiler.

In the following definitions, the symbol eop indicates a card which contains only the characters 'EOP', in columns 10-14.

4.1 SOURCE PROGRAM DEFINITION

The following definition for an ALGOL source program is based on the definition of an ALGOL program (Section 4.1.1, Chapter 2).

Syntax

<pre> ::= <empty> | <any sequence of symbols except begin, code or procedure>

<post> ::= <empty> | <any sequence of symbols except eop>

<source program> ::= <pre> <program> <post> eop

Semantics

A source program must contain declarations for all variables referenced in it. It must contain declarations for all procedures (except standard) it calls, including procedures that are compiled separately from the main program as an ALGOL source procedure (Section 5.4.6, Chapter 2).

Compilation of an ALGOL source program (generation of object code) starts with the first ALGOL symbol 'BEGIN' in the source deck and terminates with the end symbol which causes the number of begin and end symbols to be equal, or the eop card, whichever occurs first. If the eop card occurs first, however, a diagnostic is issued.

Any information in the source deck prior to the first begin or between the final end and the eop is treated as a commentary, printed as part of the source listing and included in the line count.

A program name is generated from the characters in columns 1-7 of the first source deck card, provided the character in column 1 is alphabetic. This name is terminated with the seventh character or by the first non-alphanumeric character encountered. If the character in column 1 is not alphabetic, the name generated is XXALGOL. The generated name is assigned to the subprogram output from the source program (Chapter 5) and is printed on the page headings of the source listing.

4.2 SOURCE PROCEDURE DEFINITION

The following definition of an ALGOL source procedure is based on the definition of a procedure declaration in the ALGOL-60 Revised Report (Section 5.4.1).

Syntax

<pre> ::= <empty> | <any sequence of symbols except begin, code or procedure>

<mid> ::= <empty> | <any sequence of symbols except procedure>

<post> ::= <empty> | <any sequence of symbols except eop>

<d> ::= <digit>

<code number> ::= <d> | <d><d> | <d> <d> <d> | <d><d><d><d> | <d><d><d><d><d>

<code head> ::= <pre> code <code number>; <mid>

<code tail> ::= eop | ; <post> eop

<source procedure> ::= <code head>
<procedure declaration>
<code tail>

Semantics

A source procedure must contain declarations for all variables referenced in it. It must contain declarations for all procedures (except standard) it calls, including procedures which are compiled separately as ALGOL source procedures (Section 5.4.6, Chapter 2).

A source procedure may employ the same language features as a procedure declared in a source program, except it may not be formally recursive. That is, the procedure identifier may not occur within the body of the procedure other than in a left part in an assignment statement (Section 5.4.3, Chapter 2).

Compilation of an ALGOL source procedure is initiated by the ALGOL symbol code ('CODE'). This symbol must be followed immediately by a number in the range 0-99999, and then by a semi-colon (.,). The same code number is included in the body of the declaration for this procedure in the source program or source procedure referencing it (Section 5.4.6, Chapter 2).

Compilation of an ALGOL source procedure starts with the symbol procedure ('PROCEDURE') which may be preceded by one of the type declarators real ('REAL'), integer ('INTEGER'), or Boolean ('BOOLEAN').

If the procedure symbol is encountered before the code symbol, compilation of the procedure starts normally, but an error message is issued, and the code number 00000 is supplied.

Compilation of an ALGOL source procedure ends at the normal end of the procedure declaration. If the body of the procedure is a single statement, the end is at the semi-colon (.,) terminating that statement. If the body is a compound statement or block, the end is at the semi-colon following the balance of begin and end symbols.

The semi-colon in both cases may be replaced by the eop card. If the eop card occurs before the single statement is complete or before begin and end symbols balance, a diagnostic is issued.

Information in the source deck prior to the code symbol, between the code number and the procedure symbol, and between the end of the procedure and the eop card is treated as commentary. Commentaries are printed as part of the source listing and included in the line count.

The name generated for the procedure is always CPxxxxx where xxxxx is the code number. If five digits are not specified, the number is zero-filled on the left. For example, 20 becomes 00020. Any error in the specification of the code number results in 00000. The generated name is assigned to the subprogram output from the source procedure and is also printed on the page headings of the source listing.

4.3 SOURCE INPUT RESTRICTIONS

A single source program or single source procedure, or any combination of these, may be compiled with one call of the compiler. Whether the resulting output constitutes an acceptable object program for execution depends on the mode (segmented or non-segmented) of the output, and any special binary subprogram input (Chapter 5).

The object program resulting from the compilation of a single source program in either mode, with no special binary subprogram input, is always executable, provided there are no compilation errors.

Source input for compilation must be in the form of cards or card images, described here only as cards.

Various control cards are required to request an ALGOL compilation. Included in these is the ALGOL control card (Chapter 6).

4.4 LANGUAGE CONVENTIONS

The input cards contain the character representations for the ALGOL symbols shown in Table 1. For example, to include the ALGOL symbol begin, the user punches the characters 'BEGIN'.

A blank character has no effect on the compilation process, except in strings (Chapter 2). Blanks may be freely used elsewhere to facilitate reading. For example, MEAN UPPER BOUND, MEAN UPPERBOUND, and MEANUPPERBOUND are treated as being identical (the same name). Similarly, blanks may be included in the character representation of the ALGOL symbols. The ALGOL symbol real may be punched as 'R E A L' instead of the normal 'REAL'.

4.5 CARD CONVENTIONS

Only columns 1-72 of each card are interpreted by the compiler. No syntactic meaning is attached to these boundaries; any language structure may appear across the boundaries of two or more cards.

At compile-time, each card is counted and assigned a line count (beginning at 0) for reference by error messages. This line count is included in all source language listings as are columns 73-80 of each card.

4.6 SOURCE DECK

A source deck consists of the cards which constitute one ALGOL source program or one ALGOL source procedure. The source deck must be followed by a card containing only the characters 'EOP' (eop) in columns 10-14.

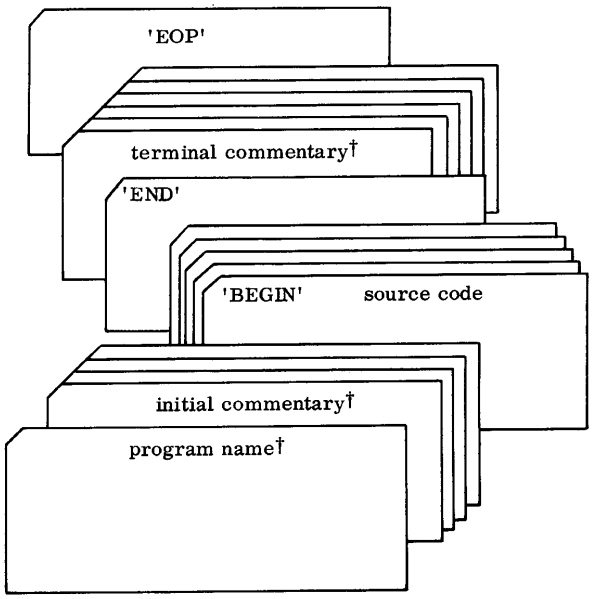
The source decks to be compiled are stacked consecutively, following the SCOPE or KRONOS control cards. The stack may contain any number of source programs or source procedures in any order within the restrictions described above. The 'EOP' card following the last source deck must be followed by a card containing only FINIS in columns 10-14. The source stack appears as one logical record on the input file. The ALGOL compiler may be called for two purposes in which no compilation of a source stack occurs:

Preparation of a segmented object program exclusively from relocatable binary subprogram input (G option)

Execution of a segmented object program which already exists from a previous compilation (R option only)

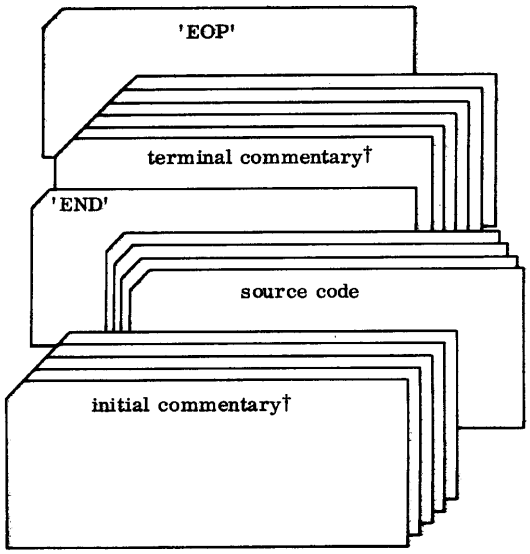
In these two cases, the source input stack and the FINIS card must be absent.

PROGRAM SOURCE DECK



† Optional

PROCEDURE SOURCE DECK



'code' nnnnn., followed by
'real' 'procedure' or
'integer' 'procedure' or
'procedure'

† Optional

Table 1. Character Representation of ALGOL Symbols

(The additional representations shown are all available with the CONTROL DATA character set, however some of the corresponding graphics do not exist in the ASCII character set. See Appendix A for differences and for the corresponding punch card codes.

ALGOL Symbol	48-Character Representation	Additional Representation	ALGOL Symbol	48-Character Representation
A-Z	A-Z		true	'TRUE'
a-z	(none)		false	'FALSE'
0-9	0-9		go to	'GO TO'
+	+		if	'IF'
-	-		then	'THEN'
×	*		else	'ELSE'
/	/		for	'FOR'
↑†	'POWER'	** or ↑	do	'DO'
÷	'/' or 'DIV'	//	step	'STEP'
>	'GREATER'	>	until	'UNTIL'
≥	'NOT LESS'	≥	while	'WHILE'
=	= or 'EQUAL'		comment	'COMMENT'
≠	'NOT EQUAL'	¬=	begin	'BEGIN'
≤	'NOT GREATER'	≤	end	'END'
<	'LESS'	<	own	'OWN'
∧	'AND'	∧	Boolean	'BOOLEAN'
∨	'OR'	∨	integer	'INTEGER'
≡	'EQUIV'	≡	real	'REAL'
¬	'NOT'	¬	array	'ARRAY'
⊃	'IMPL'		switch	'SWITCH'
.	.		procedure	'PROCEDURE'
,	,		string	'STRING'
:	..	:	label	'LABEL'
;	.,	;	value	'VALUE'
10	'	→	code ††	'CODE'
␣	␣ (space)		eop ††	'EOP'
(	(			
:=	.= or ..=	:=		
)	)			
[	(/	[		
]	/)	]		
‘	'('			
’	')'			

†In a format string, ↑ may be represented by an asterisk.

††Not defined in the ALGOL-60 Revised Report; code is defined in Section 5.4.1, Chapter 2; eop in Chapter 4.

OUTPUTS FROM COMPILATION 5

5.1 BINARY OUTPUT

The binary output (machine code) generated from one ALGOL compilation (one library call of the ALGOL compiler) may be requested in non-segmented or segmented form by a control card option.

The maximum size of the binary output generated from a single source program or source procedure deck is 131,072 words.

The non-segmented mode generates an object program which can be loaded for execution by the system loader. This mode may be requested also when the output is to be used as supplemental input to a subsequent ALGOL compilation which calls for segmented output.

When the object program and its data requirement will not fit as a whole into available memory, the segmented mode should be requested. The object program is divided into 512-word segments to be loaded by a special loader routine contained in the run-time supervisory subprogram ALG5.

5.1.1 NON-SEGMENTED OUTPUT

For each source program or source procedure deck in the source input stack (Chapter 4), the compiler generates a relocatable subprogram. These subprograms are written out on the load-and-go and/or the punch unit as specified.

After compilation, the load-and-go file also contains any subprograms written on it in the same job prior to the compilation. These subprograms may be written in any way — by an assembly, copy, or another ALGOL compilation.

An object program to be loaded for execution must contain only one subprogram generated from a source program, but it may contain the subprograms for any number of separately compiled source procedures. Any attempt to load an object program which is not legal in this sense may result in a system loader error or unpredictable execution.

Since the output from compilation need not be executed (for example, may be used only as supplementary input to another compilation), there are no compiler restrictions on the number and order of source program and source procedure decks in a source input stack (Chapter 4).

Each subprogram output by the compiler is a multiple of 512 words long and is assigned the name generated when the source deck is read.

Each subprogram contains an external name for any standard library subprograms or separately compiled source procedures called in that subprogram, and also the external name ALGORUN (the library subprogram which contains all of the global routines controlling object program execution).

Thus, a legal object program causes the loading of the object program itself, the standard library subprograms and separately compiled source procedures called, and the controlling program ALGORUN.

5.1.2 SEGMENTED OUTPUT

For an ALGOL compilation requesting segmented output, the compiler generates a segment file which contains the binary form of the object code in 512-word segments. The segmented form of the object code can be the subprograms (multiples of 512-words) of the non-segmented form, divided into individual 512-word segments. The segment file contains the subprograms for:

1. Each source program deck in the source stack
2. Each source procedure in the source stack

In addition, if supplementary user binary subprogram input is specified on the control card (U option), the file contains:

3. Each of the user's subprograms, which must be compiled in segmented mode or be coded in assembly language.

Also incorporated in the segment file are:

4. The subprograms for each of the standard library procedures called from the subprograms in 1, 2, 3 or 4
5. All remaining subprograms for each separately compiled source procedure called from the subprograms in 1, 2, 3 or 4 which are not yet incorporated

In the mode which calls for compilation from user binary subprograms (G option), the segment file does not contain the subprograms created from steps 1 and 2, since the source stack is empty.

A segment file must contain only one subprogram generated from a source program and may contain the subprograms for not more than 50 separately compiled source procedures.

A segment file may contain no more than 511 segments (261,632 words).

Since a segment file can be used only for execution under compiler control, the compiler diagnoses any infringement of these rules.

Loading and execution of an object program on a segment file is controlled by the ALG5 routine, the last pass of the compiler. Thus, a segment file can be executed immediately in the same compilation process in which it was created. Execution occurs after all source decks in the stack have been compiled and their outputs incorporated into the segment file.

A segment file may also be saved for later execution in a completely separate process. (In this case, ALG5 is called immediately, and the preceding passes of the compiler are omitted.)

ALG5 contains the global routines which control object program execution plus the segment loading routine. The segment loading routine keeps a record of each segment currently in memory. If a segment is required for execution and it is already in memory, control passes to it immediately. If not, the routine loads it from the segment file. Segments loaded are retained until available memory is full and further space is required for another segment or for the object-time stack of variables.

Segments are freely relocatable so that they may be overlaid when memory space is required. If needed again, a segment is read back into memory (though not necessarily into the same locations as previously occupied).

Object program execution requires space for at least two segments, otherwise execution cannot begin or continue normally.

Conversion of the object code from relocatable binary format (non-segmented) to segments on the segment file involves partial or total relocation of instructions before recording them on the file. This relocation takes into account the memory situation at compile-time.

5.2 ASSEMBLY-LANGUAGE OBJECT CODE

The compiler generates the object code directly into binary form, with no intermediate assembly language form. If an assembly language form of the object code is requested, the compiler encodes the binary form into COMPASS† format which may be listed or punched. The listing has the same format as a COMPASS listing, with each COMPASS instruction appearing on one line. The punch form results in a legal COMPASS assembly deck, with one COMPASS instruction punched in the proper positions in one card.

† 6000 COMPASS Reference Manual Pub. No. 60190900

5.3 SOURCE LISTING

The user may request a printed listing of any source program or source procedure compiled. Each line in the listing corresponds to one card in the source deck (one line on the ALGOL coding sheet). The lines appear in the same order as the cards in the source deck. Each line contains an exact image of the corresponding card, right shifted for readability.

Each source card in a deck is assigned a line number by the compiler, beginning at 0. Every tenth line of the listing contains the line number assigned to the corresponding card.

Diagnostics generated during compilation are printed following the source listing. Each consists of a summary of the error condition and the approximate source line number on which the error was detected.

Diagnostics are printed even if the source listing is suppressed. Chapter 8 contains a complete description of system diagnostics.

ALGOL CONTROL CARD 6

The ALGOL compiler is called from the library by the ALGOL control card.

The name ALGOL in columns 1-5 is followed by a parameter list which specifies input-output options. The parameter list is enclosed in parentheses or preceded by a comma and terminated by a period. If no parameters are specified, ALGOL must be followed by a period. The card columns following the right parenthesis or the period may be used for comments; they are ignored by ALGOL. The parameters are separated by commas and may appear in any order. Blank columns used for readability are ignored. All parameters must be fully contained on one card. The general formats of the card are:

ALGOL (c_1, c_2, c_3,...,c_n)

ALGOL, c_1, c_2, c_3,...,c_n.

Each installation may select default parameters which are assumed to be present when no conflicting parameters are given on the control card. If a control card parameter is incompatible with a default parameter setting, the default is discarded without diagnostic. If a default parameter setting needs to be de-selected, it must be done explicitly with an appropriate control card parameter, as defined below.

6.1 INPUT-OUTPUT OPTION

Each c_i has the form c or c=fn where c is any sequence of 1-7 characters beginning with one of the parameter letters defined below. For example, L and LIST are equally acceptable for the list parameter. If =fn is not specified, the standard file name associated with each parameter is used; otherwise the file name fn is used.

Except for the I parameter, the absence of any parameter suppresses the corresponding option. If I is omitted, source input is on the standard input device.

Specification of certain parameters precludes specification of others; conflicting file names are illegal. Illegal, meaningless, or contradictory combinations of parameters and/or file names are diagnosed by the compiler, which makes a legal selection, outputs the following diagnostic to the dayfile, and continues compilation normally:

ALGOL CON-CARD ERROR

v, w, x, y, z DELETED

Since any of the parameters may be pre-set as installation defaults without selection from the control card, there exists an additional form of c_i for the purpose of removing an unwanted default selection. Thus the parameter c= o has the effect of removing a possible default selection of c. As mentioned above, a default selection will automatically be removed with no diagnostic if a conflicting parameter appears on the control card.

Acceptable parameter letters are defined below.

I Source input (same as absence of I unless =fn is included). Standard file name is INPUT.

L List source input; if suppressed, only diagnostics are printed. Standard file name is OUTPUT.

X Object program in standard relocatable binary (non-segmented) load-and-go form. Standard file name is LGO.

P Object program in standard relocatable binary (non-segmented) punched form. Standard file name is PUNCHB.

S Object program in segmented form. Suppresses any X option but not P option form of the object program. Standard file name is SEGMENT. This file must be a disk file. The S option must be used when subprograms are compiled to be used subsequently by the U and G options.

R Execute the object program in segmented form. If the S option is included also, the segmented program compiled is executed. If the S option is not included, the segmented program is assumed to exist already, and all options (I, U, G in particular) are suppressed. The source stack must be completely empty. Standard file name is SEGMENT. This file must be a disk file.

U User subprogram input supplementary to I. May be included only when the S option is included. The source stack must be empty. Standard file name is LGO. Only one of the options U and G may be included.

G User subprogram input exclusively. May be included only when the S option is included. Suppresses any explicit or implicit I option. The source stack must be empty. Standard file name is LGO. Only one of the options U and G may be included.

A List the assembly language encoded form of the object code in standard assembly language listing format. Standard file name is OUTPUT.

B Punch the assembly language encoded form of the object code in standard assembly language card format. Standard file name is PUNCH.

C (Significant only when Q selected.) Perform checks on actual parameters in object program to ensure correspondence with number and kind of formals. Also perform real to integer transformations on call-by-name parameters. These checks are performed in the absence of C option except when Q is selected.

F Compile with full checking of array bounds, formal parameter specifications and real to integer transformations, if necessary, on call-by-name parameters. Absence of F implies F. This parameter is not necessary; it is supplied only for compatibility with ALGOL 2.0. It may possibly be useful protection against installation default selection of Q option for programs containing a side-effect undetectable by Q option (Chapter 2, 3.1.4.3), but this should be handled by Q = 0.

H Compile to be executed interactively, if omitted the BRKPTI statements will be ignored. (See Appendix D) The H option is not compatible with the S and R options, and it cannot be used in segmented mode. When the H option is specified, the M option also must be specified.

M Produce a map, at compile time, of source input block structure and relative stack locations of identifiers and descriptors. The map is produced, following the source listing, on the file declared by the L option or on the OUTPUT file if L is absent. Appendix E lists storage allocation requirements for this option.

Q Perform optimization of code produced for for loops and array addressing (Chapter 2, Section 3.1.4.3). Do not check array bounds in the object program. Do not check correspondence of actual parameters with formal specifications in the object program.

N Suppress array bounds checking in the object program (Chapter 2, 3.1.4.2). N option may be selected with F option.

O Optimization of generated code. The code which would be generated in absence of this option is optimized by eliminating duplicate fetches and stores and by scheduling instructions. Provides maximum utilization of X registers (all 6000 series CPU) and functional units (6600 CPU only).

D Debugging mode. Examine source program for debugging directives and install debugging code where required. If D is omitted, debugging directives in source program are ignored. (Chapter 14).

Z Syntax errors only. Compile according to other options selected unless the program has syntax errors (detected by PASS 1). If syntax errors are detected compilation terminates when the end of program is read and syntax diagnostics are listed. This provides a preprocessing mode of compilation for debugging purposes which may overcome error recovery problems.

Some typical control cards are listed below (assuming no installation defaults to be set).

ALGOL.	Compile source text from INPUT, list diagnostics only. No other output.
ALGOL, L, M, X.	Compile source text from INPUT, list source text and identifier map on OUTPUT. Produce relocatable object program on file LGO.
ALGOL, L, X, Q.	Compile source text from INPUT, list source text on OUTPUT. Produce object program, optimized for for loops and array handling, on LGO.
ALGOL, L, X, O.	Compile source text from INPUT, list source text on OUTPUT. Produce normal object code but optimized for 6000 series CPU.
ALGOL, L, Z, X.	Compile source text from INPUT, list source text on OUTPUT. If syntax errors occur then list diagnostics on OUTPUT and terminate after one pass. Otherwise continue and in absence of any other errors create relocatable object program on LGO.
ALGOL, L, S, R.	Compile source text from INPUT, list source text on OUTPUT, create object program in segmented form on file SEGMENT and execute the program in segmented mode.
ALGOL, L, D, X.	Compile source text from INPUT, list source text on OUTPUT, and create relocatable object program, including debugging code, on file LGO.

The files INTERM1, INTERM2, and LIBRARY are created by the compiler for internal use. These file names may not be used only for production programs.

6.2 COMPILATION AND EXECUTION SPEED

Compile Time

The O option increases compile time and should be used only for production programs.

The D option increases compile time and should not be used if the source program does not contain debugging directives.

The M option produces a very slight increase in compile time.

The Q option decreases compile time, particularly on programs with many for loops and subscripted variables.

Execution Time

The S option increases execution time by continually establishing the presence of program segment regardless of available field length. This option should be used only where memory requirements would not permit the whole program to be loaded.

The D option increases execution time, even when output is restored to normal by use of execution time options to suppress monitoring messages. Debugging directives should be removed as they become unnecessary. This entails recompilation.

The N option decreases execution time by removing array bounds checking.

The O option decreases execution time by eliminating superfluous instructions and rearranging code. Although intended mainly for the 6600 CPU, the modified code should execute more quickly on all 6000 series CPU's.

The Q option significantly decreases execution time. This option should be used on all production programs. Wherever possible, programs should be compiled under the Q option. Programs which cause ill behaved side effects (Chapter 2, Section 3.1.4.3) should be improved, to take advantage of the superior execution speed. If necessary the C option should be selected in addition to the Q option, and although this will reduce execution speed, the resultant reduction in execution time is still substantial.

CHANNEL CARDS 7

All input-output statements (Chapter 3) specify a channel on which the operation is to be performed and each channel is referenced by an identification number called a channel number (Section B.1.1). Each channel is associated with a set of characteristics, some of which are defined on channel cards.

Channel cards appear as the first or only cards of the object-time data on the standard input device; they are interpreted by the controlling routine before the object program is entered. The three types are: channel define, channel equate, and channel end; all must contain the characters CHANNEL, in columns 1-8.

The relationship between the structure of a file created by the input-output statements of a program and its physical representation as a file is defined by the channel card. The restrictions imposed by the operating system must be considered in creating a channel card.

7.1 CHANNEL DEFINE CARD

This card describes the characteristics to be associated with one channel number.

CHANNEL, cn=file name, Pr, PPs, Kb, Rt, M, A, Dd, B

The eight characters CHANNEL, appear in columns 1-8 followed by a list of parameters.

Each parameter describes a different characteristic. Parameters are separated by commas, and blanks may appear anywhere. The last parameter has no delimiter, but the information for one channel must be contained on a single card. Only the cn=file name parameter is required; the others are optional and may be specified in any order.

cn	channel number, unsigned integer, maximum 14 decimal digits.
file name	File name is referenced by I/O.

Pr r indicates maximum line width (r ≥ 24 characters); when omitted, P136 is assumed. If the Rt parameter has the form RnP (see below), r may be increased if necessary, from its specified value, to an exact multiple of 10 characters (whole machine words).

PPs s indicates maximum page (s lines) length. If PP0 is specified or if the parameter is omitted, no paging operations are performed. If the user defines page width or page length beyond the capabilities of the corresponding external device, data may be lost.

Kb b determines the number of consecutive blanks that serves as a delimiter for a number read or written in standard format. Omission of this parameter is equivalent to K2. The number specified must be in the range $1 \leq K \leq R$.

Rt Defines the way in which information is recorded on the external device: in this definition t can take two forms, nP and c. n and c are unsigned integers. The value r used in the following paragraph is the maximum line width specified by the Pr parameter described above. If the R parameter is omitted, R1P is assumed.

- RnP Physical records consist of n lines. Each line is r characters in length as defined in Pr.
- Rc Physical records consist of as many lines as can be wholly contained in c characters; c is increased, if necessary, from its specified value to an exact multiple of 10 characters (whole machine words). In each line trailing blanks are removed; each line ends with a record mark.
- R If t is omitted, recording characteristics are the same as for Rc form, except that the logical record length (c) has no upper bound. The end of a logical record may be defined by returning the channel to its closed state (e.g., hardware function). In this case, on input, the end of the logical record is treated as an end of file.

M Indicates input-output transmission overlap (buffering). If M is included the formatting area size (which is a function of the R parameter) is modified to allow effective overlap of the input-output transmission and the formatting processes.

A Channel is constructed without a formatting area, regardless of the inclusion of other parameters. Such a channel is usable only with the procedures GET ARRAY and PUT ARRAY. An A-type channel may be changed at object-time with the procedure CONNECT to allow it to be used with other input-output procedures.

D Meaningful only for magnetic tape. D2 sets the density to 200 bpi, D5 to 556 bpi, D8 to 800 bpi, and when D0 is used or the parameter is omitted, density is dependent on operator or installation control.

B Indicates reading or writing in binary (odd) parity; absence of this parameter sets BCD (even) parity.

7.2 CHANNEL EQUATE CARD

Channel equate cards permit the user to reference the same channel with more than one channel number:

CHANNEL, $cn_1 = cn_2$

cn_1 and cn_2 are unsigned integers with a maximum of 14 decimal digits each. Either cn_2, or a number to which cn_2 is linked by other channel equate cards, must appear on a channel define card elsewhere (though not necesarily earlier) in the set of channel cards. The channel defined on that card can be referenced by the number cn_1 as well as cn_2. Any number of channel numbers may be equated in this way with the same channel.

7.3 CHANNEL END CARD

A set of CHANNEL cards may be terminated with the following card:

CHANNEL, END

This card signifies the end of channel information as well as execution time options information, (Chapter 13). The card may be omitted except when it is necessary to separate channel information from any data appearing immediately after it on the standard input device. In the absence of this card, end-of-record, end-of-file, or non-channel information will terminate the processing of channel information. When the standard input device is a terminal, typing CHANNEL, END signifies that program execution may begin.

7.4 DUPLICATION OF CHANNEL NUMBERS

Although a channel may be associated with more than one channel number, a channel number must refer to only one channel. Therefore, the same channel number may not appear in more than one channel define card in a set. Similarly, a channel number which appears on a channel define card may not be included on the left-hand side of a channel equate card, since this is equivalent to associating that number with more than one channel.

7.5 DUPLICATION OF FILE NAMES

The following rule applies to both user-defined channels and those automatically supplied by ALGOL.

A file name may appear on any number of channel define cards; although the channels remain independent of each other, all input-output operations specifying any of the different channel numbers refer to the same file.

The same logical unit number may appear on any number of channel define cards; although the channels remain independent of each other, all input-output operations specifying any of the different channel numbers refer to the same logical unit.

7.6 STANDARD ALGOL CHANNEL CARDS

Two channel cards with standard channel numbers and characteristics are automatically supplied by the ALGOL system for the standard input and output devices as follows:

```
CHANNEL, 60 = INPUT, P80,R
CHANNEL, 61 = OUTPUT, P136, PP60,R
```

The two standard files may be referenced by the channel numbers 60 and 61 and do not require channel cards; however, these two cards are printed as part of the channel card listing as if they were specified by the user.

7.7 TYPICAL CHANNEL CARDS

Some typical channel cards are:

```
CHANNEL, 35 = NUCLEAR, P120
CHANNEL, 47 = UNCLEAR, P400
CHANNEL, 29 = 35
```

ALGOL DIAGNOSTICS 8

Three types of diagnostics are associated with the ALGOL compiler system: compiler diagnostics, compile-time and object-time I/O diagnostics, and object-time diagnostics.

8.1 COMPILER DIAGNOSTICS

Every error detected during compilation causes a diagnostic to be printed following the source listing. If the source listing is suppressed, the diagnostics are output to the standard output device. Each card of the source deck is assigned a line count, which is printed as part of the source listing, and each compiler diagnostic includes the line count of the source card in error and a brief summary of the error condition.

Compiler diagnostics are either alarms or messages; alarms cause the suppression of object code generation but messages do not.

8.1.1 COMPILER MESSAGES

The following compiler messages do not necessarily indicate the presence of an error in the source text; they provide information which may be useful in detecting errors which do not show up as language infringements:

DELIMITER IN COMMENT
LONG IDENTIFIER
NON-FORMAT STRING
PROGRAM BEGINS
PROGRAM ENDS
SOURCE DECK ENDS

PROGRAM BEGINS, PROGRAM ENDS and SOURCE DECK ENDS are output with every compilation regardless of errors.

If the line count on the PROGRAM BEGINS message is not 0, the programmer should make certain that the beginning of the program has not been treated as commentary because of a missing or misspelled delimiter (such as begin). Similarly, if the line counts on the PROGRAM ENDS and SOURCE DECK ENDS message differ, the end of the program may have been treated as terminal commentary.

8.1.2 COMPILER ALARMS

A compiler alarm indicates a serious error in the source text and suppresses object code generation regardless of user request; normal compilation and error checking continue until the end of the source text. Some errors will cause otherwise legitimate text to become invalid, but the compiler will attempt to localize the effect of each individual error. Certain errors prevent meaningful recovery and produce a secondary alarm, STOP COMPILATION; this terminates compilation, and some errors previously detected may be lost.

Some diagnostics are variations upon a fixed format and provide more information concerning the error. These diagnostics are recognized below by the appearance of one of the following structures in the message description:

<operand>
<delimiter>
<in syntax>
<line number>
<first character>
<identifier>

When such a diagnostic appears the above structure will be substituted by relevant information concerning the error, as follows:

<operand> Replaced by an identifier name (possibly truncated to nine characters), or the words A STRING or A LITERAL. If absent, operand's identity has been lost, or it is an expression containing several identifiers.

<delimiter> Replaced by an ALGOL-60 delimiter. Diagnostic representation in certain cases will be a 48-character representation so that it will be legible regardless of output medium. Hence 'NOTGR..' will represent $\leq$. The special parameter delimiter (viz Report 3.2.1.) represented by)XXX..(is included in this classification.

<in syntax> Replaced by a prepositional phrase indicating syntax expected at point of error. The following phrases may occur (numbers in parentheses indicate the relevant syntax definition as it appears in the comparison with the Revised Report in Chapter 2 of this manual):

IN EXPRESSION	(3.3.1,3.5.1.)
IN ELSE EXPRESSION	(3.3.1,3.5.1.)
AFTER ASSIGN	(4.2.1.)
IN THEN EXPRESSION	(3.3.1,3.5.1.)
AFTER ARRAY ELEMENT	(3.3.1.)
IN BOOLEAN EXPRESSION	(3.4.1.)
IN THEN STATEMENT	(4.5.1.)
IN ELSE STATEMENT	(4.5.1.)
AFTER PROCEDURE END	(5.4.1.)
AT START OF STATEMENT	(4.1.1.)

IN VALUE PART	(5.4.1.)
IN ARRAY SEGMENT	(5.4.1.)
AFTER SPECIFIER	(5.2.1.)
IN PROCEDURE HEADING	(5.4.1.)
IN TYPE LIST	(5.1.1.)
AFTER END OF FORMALS	(5.4.1.)
IN DECLARATION	(5.1.1,5.2.1.)
AFTER END OR A CALL	(4.1.1.)
IN FORMAL PARAMETERS	(5.4.1.)
IN FOR VARIABLE	(4.6.1.)
AFTER SWITCH	(5.3.1.)
IN ARRAY DECLARATION	(5.2.1.)
AFTER OWN	(5.1.1.)
AFTER FOR VARIABLE	(4.6.1.)
IN STATEMENT/DECL..	(4.1.1.)
IN SPECIFICATIONS/BODY	(5.4.1.)
IN SPECIFICATIONS	(5.4.1.)
IN OWN DECLARATION	(5.1.1.)
IN CODE NUMBER	(5.4.6.)

<line number> Replaced by a line number when the number of the line where error was detected needs to be supplemented by line number where error probably originated.

<character> Replaced by a single initial character of an unrecognizable compound delimiter. Thus, the character Q would appear for the misspelling - 'QOMMENT'.

<identifier> Replaced by first nine characters of appropriate identifier in source text.

In the following list of compilation time diagnostics corrective action is suggested. Otherwise the error is classified as implementation or field length (FL) dependent, or the programmer is referred to Chapter 2.

ARITHMETIC OVERFLOW OF NUMBER/EXPRESSION

Source text contains out-of-range number, or compiler encountered out-of-range intermediate result when evaluating an expression involving constants. (implementation restriction)

BAD PARAMETER, STANDARD PROC.. <identifier>

Compiler detected illegal parameter in call to standard procedure (procedure available without declaration). Procedure name is given. (source text error)

BLOCK REQUIRES EXCESSIVE TEMPORARIES

Too many working storage locations needed within one block. Re-order expressions, create more declared variables, or segment the block. (implementation restriction)

BYPASS OVERFLOW

Compiler capacity to stack pending forward references has been exceeded. Simplify source program. (implementation restriction)

'CODE' INTEGER REQUIRED FOR.. <identifier>

Procedure contains code body with no code integer specified. (syntax error, 5.4.6)

'CODE' LITERAL NOT INTEGER, OR BAD VALUE

Code number on this line is non-numeric, or not 0-99999 (source text error)

'COMMENT' IN ILLEGAL POSITION IN TEXT

Error in use of comment. (syntax error, 2.3)

COMMON NAME

First two characters of second COMMON not OW, for own variables. (source text for G or U option)

COMMON PRESETTING

Attempt to preset COMMON outside of defined length. (source text G or U option)

COMMON TABLE LENGTH

Length of Program Identification Table not 2 or 3. (source text for G or U option)

COMMON TEXT

Attempt to preset common area not mentioned in Program Identification Table. (source text for G or U option)

COMPILER FAILURE (PASS 2)

This diagnostic should not appear; indicates an internal inconsistency in Pass 2 of compiler.

COMPILER STACK FAILURE (PASS 3)

This diagnostic should not appear; indicates an internal inconsistency in Pass 3 of compiler.

DATA NAME/LENGTH

First COMMON name is not DATA or length is not consistent with previous DATA length. (source text for G or U options)

DECLARATION MISSING FOR.. <identifier>

Specified identifier is undeclared. (source text error)

DECLARATION AND/OR FOR-LOOPS EXCEED FL

Increase field length to permit processing of nested blocks and for-loops.

DEFERRED CODE CAPACITY EXCEEDED

In optimizing mode, compiler stacks invariant code to move to less frequently executed areas. Capacity of stack has been reached by continual deferment of such code. Disable optimization or restructure program. (implementation restriction)

DELIMITER REQUIRED PROCEEDING <operand>

Specified operand is second of two consecutive operands with no intervening delimiter. (syntax error)

DOUBLE DECLARATION OF.. <identifier>

Specified identifier, which appears in a declaration on this line, has been declared subsequently in same block. (source text error)

DOUBLE DEFINED

Two or more separately compiled procedures with same name found during preparation of segment file. (source input for S option)

DOUBLE SPECIFICATION OF.. <identifier>

Formal parameter appeared in more than one specification in this procedure heading. (source text error, 5.4.5)

DYNAMIC BOUNDS ILLEGAL FOR 'OWN' ARRAY

Dynamic own arrays are not implemented. (implementation restriction)

'END'S MISSING

More begin than end symbols found when eop encountered. (source text error 4.1.1)

EXCESSIVE NESTING OF 'BEGIN' STRUCTURES

Compiler allows combined nesting of blocks and compound statements to depth of 96. Limit has been exceeded. Simplify program structure (implementation restriction.)

EXCESSIVE NUMBER OF UNIQUE IDENTIFIERS

Compiler allows 3583 unique identifiers. Limit has been exceeded. Use same spelling for identifiers in different blocks. (implementation restriction)

EXTERNAL REFERENCE ADDRESS

Address of external reference outside current segment. (source text for G or U option)

EXTERNAL REFERENCE RELOCATION

External references may occur only from program part. (source text for G or U option)

EXTERNAL STACK OVERFLOW

More than 50 separately compiled procedures occurred during preparation of segment file. Disable S-mode or combine pre-compiled procedures. (implementation restriction)

FILL ADDRESS

Address of common reference outside of current segment. (source text for G or U option)

FILL RELOCATION

Attempt to make common reference from negative program relocatable part or from COMMON not mentioned in Program Identification Table. (source text for G or U option)

<delimiter> FOLLOWS AN INCORRECT OPERAND

Illegal operand preceeds delimiter. (syntax error)

<delimiter> ILLEGAL < in syntax>

Delimiter is illegal in stated context. Check list of possible contexts for syntax references. (syntax error)

ILLEGAL ACTUAL PARAMETER TO.. < identifier>

Procedure is called on this line with an undeclared or untyped actual parameter.

ILLEGAL CONTROL VARIABLE IN IMPLIED-FOR

Control-variable of implied-for loop is not identifier. (syntax error)

ILLEGAL LOCAL ARRAY-BOUND.. < identifier>

Identifier appears in bounds of an array and is declared in same block. (source text error, 5.2.4.2)

ILLEGAL 'VALUE' SPECIFICATION.. < identifier>

Formal parameter specification does not permit a value. (source text error, 4.7.3.1)

INCOMPLETE ENTRY TABLE

Incorrect Entry Point Table. (source text input to G or U option)

INCOMPLETE LINK TABLE

Incorrect Linkage Table. (source text input to G or U option)

INCOMPLETE REPLICATION TABLE

Incorrect Replication Table. (source text input to G or U option)

INCOMPLETE TRANSFER TABLE

Incorrect Transfer Table. (source text input to G or U option)

INCONSISTENT CONDITIONAL-EXPRESSION TYPE

Expressions following then and else in if expression differ in type. (source text error)

INCONSISTENT SUBSCRIPT COUNT.. <identifier>

Array appears in reference in which subscript count differs from number of subscripts in declaration, or, for formal array, from previous subscript count. (source text error)

INCONSISTENT TYPES WITHIN AN EXPRESSION

Variation in types of component elements of general expression on this line. (source text error)

INCORRECT EXPRESSION TYPE IN 'WHILE'

In for list element containing while, preceeding expression is not arithmetic, or following expression is not Boolean. (source text error, 4.6.1)

INCORRECT LABEL USAGE OF.. <identifier>

Identifier used as label has been declared differently. (source text error)

INSUFFICIENT FL FOR COMPLEX STATEMENT

Compiler requires more memory to generate code for this statement at this point. Request more FL, or simplify statement or program nesting. (implementation restriction)

INSUFFICIENT MEMORY TO SAVE DECLARATIONS

Program structure exceeds compiler's capacity to stack declarations of outer block to process new declarations. Request more FL or ensure that procedure declarations occur as late as possible in a block head. (implementation restriction)

INSTRUCTION/CONSTANT OVERLAP

This diagnostic should not occur; it is a double check to ensure generated code will fit into a segment. It indicates a compiler failure in Pass 4.

LITERAL AFTER 'CODE' IS NOT AN INTEGER

Non-numeric character found after code in the code head part of source text. (source text error)

LOAD ADDRESS

Load address in text table is out of range. (source input to G or U option)

MAXIMUM LEVEL OF BLOCK-NESTING EXCEEDED

A block nested to a level greater than 32. Simplify block structure. (implementation restriction)

MISSING PROGRAM OR 'BEGIN' UNDETECTED

End-of-record or 'EOP' encountered before appearance of program. Check hardware representation of delimiters. (source text error)

MORE THAN ONE PROGRAM

More than one main program occurred during preparation of segment file. (error in input to S-mode)

NEGATIVE, OR ZERO, ARRAY SIZE IMPLIED

Array size compiled from constant bounds is negative or less than zero. (source text error)

NEW SEGMENT WITHIN TEXT TABLE

Text Table overlaps two segments. (error in input to G or U option)

NO ARRAY/SWITCH DECLARATION.. <identifier>

Identifier used as array or switch without being declared as such. (source text error)

NO MAIN PROGRAM

Only code procedures occurred during segmentation. (error in input to S-mode)

NO PROCEDURE DECLARATION FOR.. <identifier>

Identifier used as procedure without being declared as such. (source text error)

NON-ARITHMETIC ARRAY-BOUND EXPRESSION

(source text error)

NON-ARITHMETIC 'FOR' CONTROL VARIABLE

For control variable on this line not a simple or subscripted arithmetic variable. (source text error)

NON-ARITHMETIC SIMPLE 'FOR' LIST ELEMENT

(source text error, 4.6.1)

NON-ARITHMETIC 'STEP' ELEMENT

(source text error, 4.6.1)

NON-ARITHMETIC SUBSCRIPT FOR.. <identifier>

Array has non-arithmetic subscript. (source text error)

NON-BOOLEAN EXPRESSION IN AN 'IF' -CLAUSE

Expression following if symbol not Boolean. (source text error)

NON-DESIGNATIONAL EXPRESSION IN SWITCH

Element in switch list on this line not a label or a designational expression. (source text error)

NUMBER OF UNIQUE IDENTIFIERS EXCEEDS FL

Symbol table overflowed available memory. Request more FL or use same spelling for identifiers in different blocks. (implementation restriction)

NUMBER SYNTAX ERROR

Number in source text is incorrectly formed. (syntax error, 2.5.1)

OPERAND OVERFLOW IN COMPLEX STATEMENT

Complicated statement exceeded capacity of compiler to stack operands. Re-order expressions or split into intermediate statements. (implementation restriction)

OPERATOR OVERFLOW IN COMPLEX STRUCTURE

Complex program structure exceeded capacity of compiler to stack operators. Simplify immediate statements or surrounding program structure. (implementation restriction)

PARAMETER COMMENT INCORRECTLY FORMED

Incorrect special parameter delimiter replaces comma in parameter list. (syntax error, 4.7.1)

PARAMETER COUNT ERROR IN.. <identifier>

Number of actual parameters in call to procedure does not equal number of formal parameters in declaration. (source text error)

RECURSIVE PRE-COMPILED PROCEDURE

Separately compiled procedure may not be recursive. (implementation restriction)

REFERENCE OUTSIDE SEGMENT

Invalid addressing encountered during preparation of segment file. (error in input to S-option)

REPLICATION ADDRESS

Attempt to perform replication outside current segment. (error in input to G or U option)

REPLICATION RELOCATION

Replication may occur only within program part. (error in input to G or U option)

<delimiter> REQUIRES A PRECEDING OPERAND

Delimiter is second of two consecutive operators with no intervening operand. (syntax error)

SEQUENCE

Binary input out of sequence. (error in input to G or U option)

SPECIFICATION REQUIRED FOR.. <identifier>

Above format mentioned in specification. Omission of specifications not permitted. (implementation restriction, 5.4.5)

SPECIFIED FORMAL MISSING.. <identifier>

Identifier appeared in specification part but not in formal parameter part of procedure declaration. (source text error)

STACK OVERFLOW IN OPTIMIZING FOR LOOPS

Optimizing process which moves invariant code to less frequently executed parts of program has reached capacity. Disable optimization mode or perform source optimization (e.g. centralization of subscript expressions) by hand.

STOP COMPILATION

Fatal error occurred; (e.g. a stack overflow) no further diagnosis will take place.

STRING LENGTH ERROR, ORIGIN AT LINE <line number>

Number of characters in string exceeded maximum at line number. Line number in message indicates where string started. Ensure string quotes are correctly represented. (source text error)

<delimiter> STRUCTURE ENDED BY <delimiter>

Second delimiter has been encountered without prior occurrence of a pairing delimiter. Either might be in error, e.g. A(B] (syntax error)

SUBPROGRAM SIZE

Size of current subprogram exceeds maximum size of target machines. (implementation restriction)

THIS IS SECOND DECLARATION OF.. <identifier>

Identifier appeared in previous declaration at line number given in corresponding DOUBLE DECLARATION diagnostic. (source text error)

TOO MANY ACTUAL PARAMETERS IN <identifier>

Procedure called with more than 63 actual parameters. (implementation restriction)

TOO MANY LOCAL VARIABLES IN THIS BLOCK

Maximum number of local variables allowed in single block exceeded; declare variables in different blocks. (implementation restriction)

TOO MANY SUBSCRIPTS/ELEMENTS.. <identifier>

Array or switch has more than 63 subscripts or elements. (implementation restriction)

UNDEFINED

No external name found during preparation of segment file. (error in input to G or U option)

UNRECOGNIZABLE COMPOUND DELIMITER ' <character> ...

Structure with escape and specified character as first two characters not recognizable as compound delimiter. Check spelling of delimiter or incorrect usage of escape character. (source text error)

UNTERMINATED STRING STARTS AT LINE <line number>

End of record or 'EOP' encountered before terminal string quote of string starting at specified line. Check for incorrect representation of string quote. (source text error)

X-REGISTER ASSIGNMENT FAILURE

Pass 4 code optimizer is unable to proceed with register allocation in a complex program structure. Simplify complex statements at the point of failure (e.g. multi-line statements with calls to standard procedures) or remove the O-option.

8.2 COMPILE-TIME AND OBJECT-TIME I/O DIAGNOSTICS

System diagnostics concerning input-output usage at compile-time and object-time appear in DAYFILE:

ALGOL-I/O-ERROR yyy ON FILE file-name

yyy values:

P_1	Record length error in GETARRAY	object-time only
P_3	Attempt to close OUTPUT, PUNCH or PUNCHB files	
P_4	Attempt to close INPUT file	

In general, the following values of yyy result from system error, improper use of I/O system in a handwritten procedure, or wrong segment file:

yyy		
FC1	Illegal function code to I/O	compile-time or object-time
FC2	Error on call for open formatted	
FC3	Error on call for close formatted	
FC4	Error on call for read or write formatted	
FC5	Error on call for function non-formatted	
FC6	Error on call for read or write non-formatted	
FC7	Illegal status returned after physical I/O	
SG1	Error in the segment file	object-time only
SG2	Library routine missing from segment file	
INT	Error in compiler inter-pass I/O	compile-time only

8.3 OBJECT-TIME DIAGNOSTICS

Upon normal exit from an object program, the contents of all non-empty format areas are output. The following message printed on the standard output device indicates a successful execution:

```
END OF ALGOL RUN
```

Upon abnormal termination, a diagnostic is printed on the standard output device to indicate the nature of the error, and the contents of all non-empty format areas are output. Information which traces the execution path through the currently active block structure is then printed on the standard output device as follows:

```
THIS ERROR OCCURRED AFTER LINE xxxx
IN THE BLOCK ENTERED AT LINE xxxx
     (global stack information)
     (local stack information)
THIS BLOCK WAS CALLED FROM LINE xxxx
IN THE BLOCK ENTERED AT LINE xxxx
     (local stack information)
THIS BLOCK WAS CALLED FROM LINE xxxx
```

Stack information for each block is printed following the corresponding BLOCK ENTERED line.

Object-Time Diagnostics

ALPHA FORMAT ERROR

Output value is too large.

ARITHMETIC OVERFLOW

Evaluation of an expression results in arithmetic overflow (e.g., division by zero) for which no provision has been made with ARTHOFLW procedure.

ARRAY BOUNDS ERROR

Computed element address in an array is not within total array boundaries.

ARRAY DECLARE ERROR

Computed array size is negative or zero.

ARRAY DIMENSION ERROR

Number of dimensions in actual parameter array in procedure call differs from number in formal parameter array.

ARRAY HAS MORE THAN 20 SUBSCRIPTS

Excess subscripts are not accepted.

ARRAY xxx HAS nnn DIMENSION(S)

Subscript count is incorrect.

BLOCK LEVEL OUT OF RANGE

Occurs after a B = n option; the block level n is either negative or greater than 32. Check n against the block level in the map.

BOOLEAN INPUT ERROR

In Boolean formats F or P, input character is not F or T, or 0 or 1, respectively.

CHANNEL xxxxxxxxxxxxxx

Defines channel on which preceding error occurred.

CHN xxxxxxxxxxxxxx

Defines channel on which preceding error occurred.

CHANNEL CARD SYNTAX - CIRCULAR - PARITY - EOF

Syntax of channel card is wrong or define and equate cards result in a circular definition of a channel; or an irrecoverable parity error or EOF card occurred during reading of channel cards. Incorrect card is output before program terminates.

DISPLAY EXCEEDED

Block structure is nested to more than 32 levels; this error can occur only becuase of calls to separately compiled procedures.

ELEMENT OF xxx OUT OF BOUNDS

Subscript values indicate an element outside of the array bounds.

EXPONENTIAL ERROR

Argument of EXP procedure is too large.

FORMAT ITEM ERROR

More characters in expanded format item than permitted in INPUT, OUTPUT, IN LIST and OUT LIST.

FORMAT MISMATCH

Syntactically correct format string appears incorrect (probable machine or system malfunction).

FORMAT REPLICATOR

Replicator in call to FORMAT procedure out of range.

FORMAT STRING ERROR

Incorrect format string.

GET/PUT ARRAY ERROR

GET ARRAY and PUT ARRAY may not be used when formatting and format area have been specified for channel.

'GO' MUST BE FOLLOWED BY CR

Anything following 'GO' is ignored; retype 'GO' to restart program execution.

H/V LIM ERROR

H LIM and V LIM arguments L, R and L', R' out of range.

IDENTIFIER...NOT DECLARED

This identifier never has been declared in this program.

INCORRECT NUMBER FORMAT AFTER

after B = or X = the number must be an integer.

In an assignment to a variable, the number format must correspond to the variable type since there is no type conversion.

ILLEGAL IN-OUT

Illegal operation requested for equipment selected.

ILLEGAL MODE CALL

T parameter in call to MODE procedure out of range.

ILLEGAL STRING INPUT

Attempt to read into a string parameter during a call of INPUT or IN LIST.

I/O CHANNEL ERROR

Normal input-output procedures, except GET ARRAY and PUT ARRAY, cannot be performed on non-formatted channels.

xxx IS NOT AN ARRAY IDENTIFIER

Identifier before [is not an array identifier.

xxx IS NOT A BOOLEAN VALUE

A Boolean variable will accept only 'TRUE' or 'FALSE'

LAYOUT CALL ERROR

Procedures established by H END, V END, and label set by NODATA are not accessible after return from layout procedure called by IN LIST or OUT LIST.

LOGARITHM ERROR

Argument to LN procedure may not be negative or zero.

NO VALUE GIVEN FOR

This identifier appears in the program, but it has no value accessible at this point: it may be a formal parameter; it is not an arithmetic or Boolean variable; or it is not accessible from the current block.

NON-FORMAT INPUT ERROR

In non-formats, I, R, L, or M, input field contains non-octal characters.

NUMBER SYNTAX ERROR

Number input in standard format does not conform to proper syntax.

NUMERIC INPUT ERROR

Data input under format control does not conform to numeric input format.

OUT CHARACTER ERROR

Parameter OUT CHARACTER call is not in proper range.

PARAMETER COUNT ERROR

Number of actual parameters in procedure call incorrect.

PARAMETER KIND ERROR

Kind of actual parameter in procedure call does not correspond to kind of associated formal parameter.

PARAMETER TYPE ERROR

Types of actual and formal parameters in procedure call do not correspond.

PROCEDURE SELECTION EXCEEDS LIMIT

No more than 10 procedure names can be selected through the option P = .

S-UNIT ILL. ON CHANCARD

Unit defined on channel card is same as unit containing segment file.

SIN - COS ERROR

Argument to SIN or COS procedure is too large.

SQUARE ROOT ERROR

Argument to the SQRT procedure may not be negative.

STACK OVERFLOW

Data requirements of program exceed available memory.

STANDARD OUTPUT ERROR

Standard output can be used only for numeric and string formats.

STRING ELEMENT ERROR

Rules of STRING ELEMENT violated.

SWITCH BOUNDS ERROR

Value of switch designator out of range.

SYNTAX ERR AFTER P =

If the given name has more than ten characters, the tenth character must be replaced by an arrow (⌐→).

SYSPARAM-CHANNEL

SYSPARAM procedure can be called only for formatted channels.

SYSPARAM - WRONG F

SYSPARAM called with incorrect F parameter.

SYSPARAM - WRONG Q

SYSPARAM called with incorrect Q parameter.

TABULATION ERROR

Argument of TABULATION not in proper range.

THIS PROCEDURE NAME IS ON THE LIST

Warning message; the procedure was requested more than once.

UNASSIGNED CHANNEL

No channel defined for channel number used in program.

UNCHECKED EOF

End-of-file mark detected, but no provision made with EOF procedure.

UNCHECKED PARITY

No PARITY procedure for parity error detected.

UNDEFINED DSI

A file is not opened for the specified dsi.

UNDEFINED FOR LABEL

Attempt to jump into middle of for statement.

xxxxx UNRECOGNIZABLE. ALL THAT FOLLOWS HAS BEEN IGNORED.

x probably is followed by a wrong separator character or the identifier x has more than 10 characters.

COMPILER DESCRIPTION 9

The compiler consists of the subprograms ALGOL, ALG0, ALG1, ALG2, ALG3, ALG4, and ALG5. Only ALG1 through ALG4 take part in the actual translation of a source text into object code. ALGOL controls the compilation process, ALG0 handles the control card, and ALG5 controls segmented object program execution.

ALG1, ALG2, ALG3 and ALG4, each perform a separate function in the translation process as described below.

9.1 INFORMATION FLOW

The output from each pass of the compiler is in the form of bytes representing an internal form of the source text. The output stream of bytes from one pass serves as the input stream to the next pass.

The bytes generated by each pass are first stored in available memory. If the entire stream fits there, it is retained for processing by the next pass. Otherwise, it is written out as a scratch file which is read back in by the next pass and the output from that pass is written onto a second scratch file.

Each pass processes the stream of bytes in a direction opposite to the one in which they were generated by the previous pass: ALG1 and ALG3 process the bytes in the source text order; ALG2 and ALG4 process them in the reverse order.

9.2 LANGUAGE TRANSLATION

Each pass uses one or more internal pre-set tables; the value of each byte input to a pass is an index into a table in that pass. The table entry thus referenced either yields an output byte value for the next pass or it directs control to an action which will generate an output byte value. One or more input byte values may generate one or more output byte values.

9.3 LANGUAGE ANALYSIS

The syntactic analysis in ALG1 uses a state/delimiter method. The delimiter is the current input byte which represents an element in the language. The state is the current syntactic situation, as established by previous delimiters; each delimiter causes a change in state.

A pre-set table in ALG1 contains the syntactic rules of the ALGOL language. This table is referenced by the two dimensions state and delimiter. Each entry in the table, referenced in this manner, indicates whether or not the current delimiter is legal in the current state, the new state to be established, and any further actions required.

Similar table actions are performed in other passes to check different aspects of a program for legality.

9.4 IDENTIFIER (SYMBOL) TABLE

The identifier table is established by ALG1 as it reads the source text. The table contains only one version of each distinct identifier regardless of the number of times that identifier occurs in the source.

Each identifier in the table is linked to the next identifier with the same classification. The classification of an identifier is derived from the hash-total of the characters comprising the identifier. Whenever the identifier table is searched, only those identifiers on the classification chain corresponding to the one being considered are examined.

Each distinct identifier is associated with an integer from 513 to 4095 assigned sequentially as distinct identifiers are encountered. The integer values are output as the byte values for the identifier for processing by subsequent passes. The identifier table is not used after the first pass of compilation.

9.5 COMPILER SUBPROGRAM DESCRIPTIONS

ALGOL

The ALGOL subprogram is the internal controller of the compiler. It is loaded from the library as a result of a control card specifying the name ALGOL. Its main function is to load and pass control to each pass of the compiler as required. At the end of each pass, control is returned to ALGOL for loading the next pass. At the end of compilation, ALGOL returns control to the operating system. ALGOL resides in memory throughout the compilation process.

ALG0

ALG0 processes the control card parameters. It checks specified options for legality and sets the appropriate information for these options for later use.

ALG1

ALG1 reads the source code and translates it into an internal form of the ALGOL-60 reference language. It performs all syntax analysis on the source program, generates the output stream of intermediate information for processing by ALG2, and prints the source text in parallel with the processing.

ALG1 consists of various control routines plus the three routines TASK1, TASK2, and TASK3 which perform the principal functions. TASK3 is immediately called on entry to ALG1; TASK3 references TASK2 to perform subsidiary functions; TASK2 itself calls TASK1 for functions subsidiary to it.

TASK1 analyzes and checks the hardware representation of the source text, and converts it to the internal form of ALGOL symbols. All comment structures are removed.

TASK1 also assembles strings and transmits these and any diagnostic alarm indications directly to the output stream for later processing by ALG2. TASK1 is called by TASK2 whenever the latter requires another ALGOL symbol.

TASK2 processes the intermediate bytes from TASK1. It assembles identifiers into a table, and for each distinct identifier, it outputs a unique integer value byte to TASK3. TASK2 also assembles numeric constants and outputs a byte for each to TASK3. It transfers information concerning each constant and any alarm indications directly to the output stream. TASK2 is called by TASK3 when the latter requires the intermediate byte describing the next source text element.

TASK3, using the bytes furnished by TASK2, examines the entire program for syntactic correctness. It also rearranges the procedure headings for more convenient processing in later passes and further classifies certain delimiters, such as the comma and semi-colon, into more exact contextual meanings. The results of this processing are transferred to the output stream for processing by ALG2.

If an error is found in a structure, TASK3 suppresses the remainder of the structure up to the terminating delimiter (such as ; , end, then, else, etc.) replacing it with the special byte which indicates trouble and bytes indicating the nature of the error. (ALG2 processes this information in reverse order; it encounters the trouble byte and then removes that part of the structure which was already in the output stream before TASK3 detected the error.)

ALG2

ALG2 performs two principal tasks:

It detects each declaration or declaration-like entity, and develops a systematic representation for them. This representation is output for processing by ALG3 directly behind the begin of the appropriate block.

It totally removes from the output stream statements marked by the trouble byte. In general, this ensures syntactic correctness of the remaining code.

ALG2 processes the input stream of bytes in the reverse of the source text order (opposite direction to ALG1 output). ALG2 consists of various control routines plus the routine TASK4 which performs the two principal functions.

ALG3

ALG3 performs the following major functions:

Generates relative addresses of all variables in the stack.

Checks kinds and types to ensure consistency between the declaration of a given variable and its use in the statements and expressions of the program.

Generates machine code in a macro-like format which is convenient for ALG4 to use in the final generation of program addresses.

ALG3 processes the input stream of bytes in the source text order (opposite order to ALG2 output). ALG3 consists of various control routines plus the three routines TASK5, TASK6, and TASK7 which perform the principal functions.

TASK5 processes the declarations of a block; it sets up a declaration table which contains descriptions of all currently accessible variables. Each description includes the kind, type, block level, and block relative address of the associated variable. This table is referenced by TASK6.

TASK6 checks the types and kinds of all variables for consistency between declaration and use in the program; and it sets up the reverse Polish notation of the source text, calling upon TASK7 for each element of this Polish string. An error-free program is assumed in TASK6 (as far as delimiter structure is concerned), this situation having been achieved by TASK3 and TASK4.

TASK6 contains three sections: it handles operands as they are encountered from the input; it handles incoming operators and stacks them if necessary; and it handles the priority processing of stacked operators.

TASK7 is called by TASK6 for each element of a Polish string. Each operand element is stacked by TASK7 until it encounters an operator element. At this time, it generates the object code (in macro-like format) for the operator and the corresponding stacked operands.

After each call to TASK7 (for either an operand or operator), control returns to TASK6 for the next element in the Polish string.

ALG4

ALG4 generates all output from the compiler. The major outputs are: assembly form of object program (options A and B), relocatable binary form of object program (options X and P), and segmented form of object program (option S).

ALG4 processes the input stream of bytes in the reverse of the source text order (opposite direction to ALG3 output).

ALG4 consists of various control routines plus the three routines TASK8, TASK9, and TASK10 which perform the principal functions. TASK8 handles the input bytes from ALG3; TASK9 handles binary relocatable input, and TASK10 generates all of the compiler outputs (except for the source listing) from information furnished by TASK8 and TASK9.

TASK8 generates units of 512 machine instructions and constants from the macro-like bytes output by ALG3. TASK8 scans backwards through the translated program, generating code from the last end to the first begin, filling in final program point addresses.

TASK8 builds up each unit in a 512-word area, stacking instructions from the end with higher address towards the end with lower address, and stacking constants from the other end. When the two meet, TASK8 calls TASK10 to output the unit.

TASK9 converts subprograms from a relocatable binary form into units of 512 words as described for TASK8. Like TASK8, it calls TASK10 for every such unit prepared.

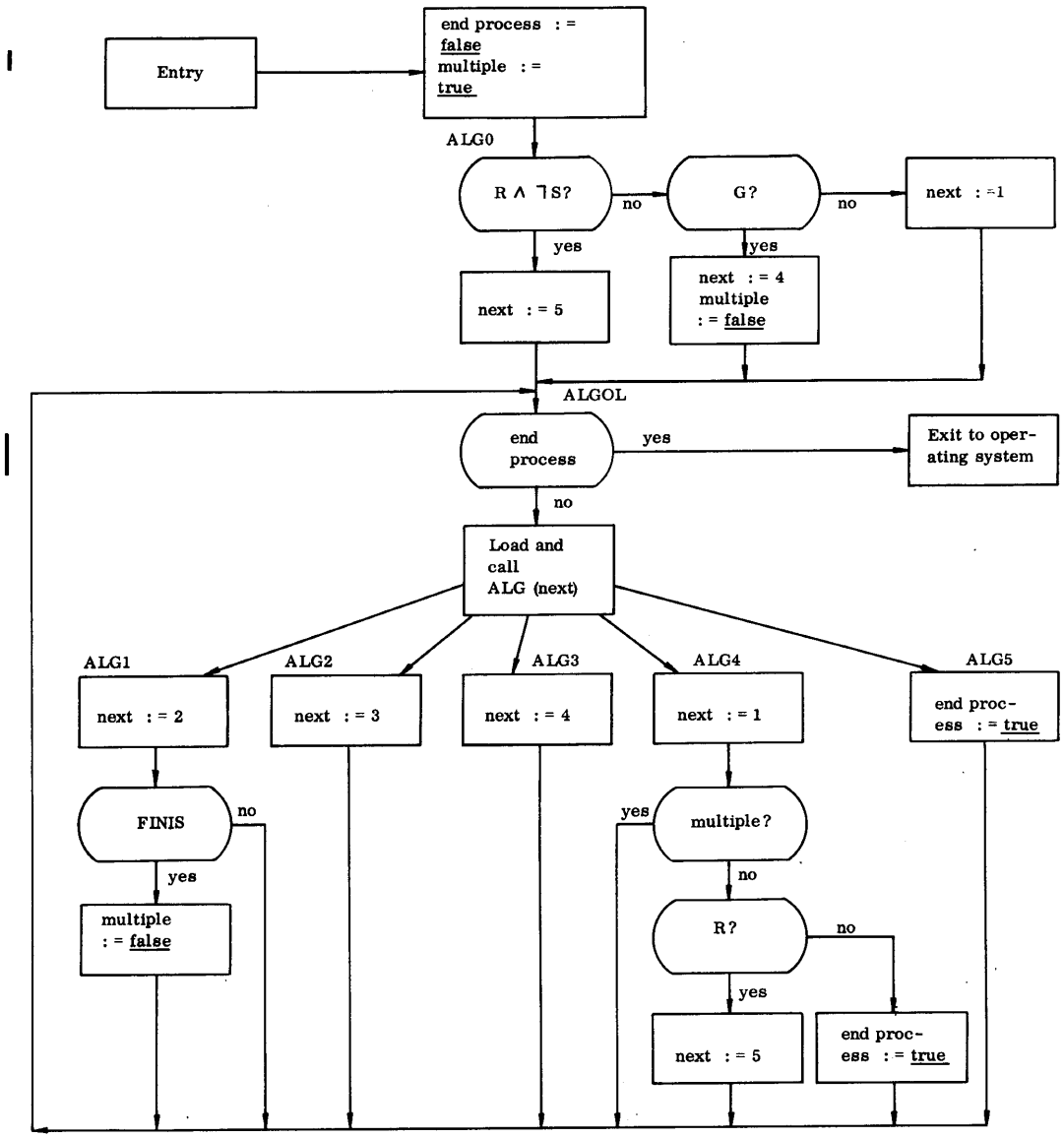
TASK10 is called by both TASK8 and TASK9 when either has a complete unit of 512 instructions and constants to output. The three main sections of TASK10 handle assembly language output (A and B options), relocatable binary output (X and P options), and segmented output (S option). Depending on the outputs requested, any or all of these sections are executed for any one unit.

9.6 OVERALL COMPILER FLOW

Not all of the passes ALG1 through ALG5 are loaded and executed for every compilation requested. The selection of passes is based on the compilation and object-code output options specified on the control card (Chapter 6). Each pass sets information indicating the next pass to be loaded.

The following diagram illustrates the overall flow of the compiler passes based on the control card options specified.

OVERALL FLOW DIAGRAM OF ALGOL COMPILER



OBJECT PROGRAM 10

10.1 RUN-TIME SUPERVISORY PROGRAM

A run-time supervisory program, called ALGORUN for non-segmented execution and ALG5 for segmented execution, controls object program execution in either segmented or non-segmented form. The supervisory program consists of controlling routines which handle the dynamic stack of variables and the segment structure. ALG5 and ALGORUN are functionally identical except that ALG5 additionally contains the segment loading and controlling routine.

All calls to the separate global routines are generated within the object program as machine jumps to different positions in a data vector. Each position of the data vector contains the address of one particular routine at execution time. The contents of the data vector are defined by the supervisory program.

10.2 OBJECT-CODE STRUCTURE

Regardless of the final form of the output requested, the object code is generated in segments of 512 machine words. Each segment consists of instructions and any constants referenced by these instructions. The object code is generated backward (from the last to the first begin).

Within each segment, the constants are stacked above the instructions and are given lower addresses than the instructions. The instructions are sequenced so that execution within each segment proceeds normally from low address to high. The last instruction in each segment is a system jump to the first instruction of the next logical segment (the one physically preceding). The program is entered at the last segment generated (the logical beginning of the program).

10.3 OBJECT-CODE GENERATION

The object code is generated first in an internal representation of the final code in segments of 512 words as described above. The non-segmented output is obtained directly from this representation; the segments collectively form a relocatable binary subprogram. The segmented output is obtained by modifying the address fields of instructions so that each segment is individually relocatable; each modified segment is output separately to the segment file. (A binary subprogram which already physically exists is incorporated into a segment file by first converting it to the internal representation.) It, therefore, follows that the two forms of the object code are structurally identical.

10.4 LIBRARY SUBPROGRAMS

The library subprograms obey the same structural rules as a binary subprogram generated by the compiler. Each subprogram consists of one or more segments, and each contains one or more of the standard procedures. The standard procedures are organized logically, so that functionally similar procedures (such as SIN and COS, and IN REAL and IN ARRAY) are contained in the same subprogram.

10.5 ADDRESS-FIELD CONVENTIONS

Three basic types of references can be made in an object program: a reference to the stack, and a reference from one instruction to a point in the same segment, or to a point in a different segment. Correspondingly, three types of address fields are generated. In addition, except for stack references, the forms of the address fields are different in the non-segmented and segmented output, though at execution time (after loading), they are essentially the same.

Reference to the Stack

Every variable in the stack is referenced relative to a stack reference address (Chapter 11), which is the beginning address of the stack area reserved for the block in which the variable is declared. Such references are therefore generated as the relative number (position) of the variable in its block plus an index register which contains the appropriate stack reference address.

Reference to the Same Segment

Such a reference can be either a reference to a constant in the segment or a compiler-generated jump to a point in the same segment (such as a bypass in an if statement).

In the non-segmented form, the address field is a normal subprogram address, relative to the beginning of the whole subprogram. In the binary form, this field is flagged to indicate that it requires positive program relocation by the system loader, which results in the desired absolute address.

In the segmented form, the address field is changed to the relative position of the desired location within the segment itself (000YYY). This field is flagged in the segment file to indicate that it requires segment relocation when the segment is loaded, which results in the desired absolute address.

Program jumps (as opposed to compiler-generated jumps), such as those generated from go to statements, are generated in the form of a reference to a different segment even if the destination address is in the same segment.

Reference to a Different Segment

Such a reference can result only from a transfer of control requirement (which may necessitate a jump out of the segment). Because of this, all such transfers are performed through the run-time supervisory subprogram, in both the segmented and non-segmented forms.

This transfer of control is generated as a call to one of the routines in the supervisory subprogram, with the destination address as a parameter to call.

In the non-segmented form, the address field is the complement of the normal subprogram address in relation to the beginning of the whole subprogram. In the binary form, this field is flagged to indicate that it requires negative program relocation by the system loader, which results in the complement of the desired absolute address.

In the segmented form, the address field is changed to MMMYYY where MMM is the segment number of the referenced segment on the segment file. This is interpreted by the controlling routine as relative location 000YYY within the segment MMM (which is loaded if not already available).

A reference from the program to a library or separately compiled procedure has exactly the same form as a reference to a different segment, except for the form of the destination address in the actual program. This always has the form 000YYY where YYY is a constant which is used as a relative address into a table which contains the addresses of all such procedures.

OBJECT-TIME STACK 11

11.1 STACK STRUCTURE

According to the rules of the ALGOL language, a variable is active (available for reference) in any block to which it is local or global. A variable is local to the block in which it is declared and global to the sub-blocks within the block in which it is declared.

Depending on the block structure and the variables declared at each level, not all variables are active at the same time. The object programs produced by ALGOL overlay variables which are not simultaneously active. The overlay process is described below.

During execution of an object program, all variables are contained in a variable-length memory stack consisting of 60-bit entries, one or more pertaining to each active variable. Since the stack includes only active entries, the size fluctuates.

The compiler assigns to each variable an address relative to the stack reference for the block in which that variable is declared in the reverse order of their declarations. The stack reference for each block is the position in the stack where the entries for that block are assigned at object-time. It is derived as follows:

When a new block is entered which is nested in the last block entered, the stack reference for the new block is assigned to the first available (inactive) position in the stack. Certain preliminary information, including the stack reference of the next outermost block, is set into the stack, beginning at this reference point.

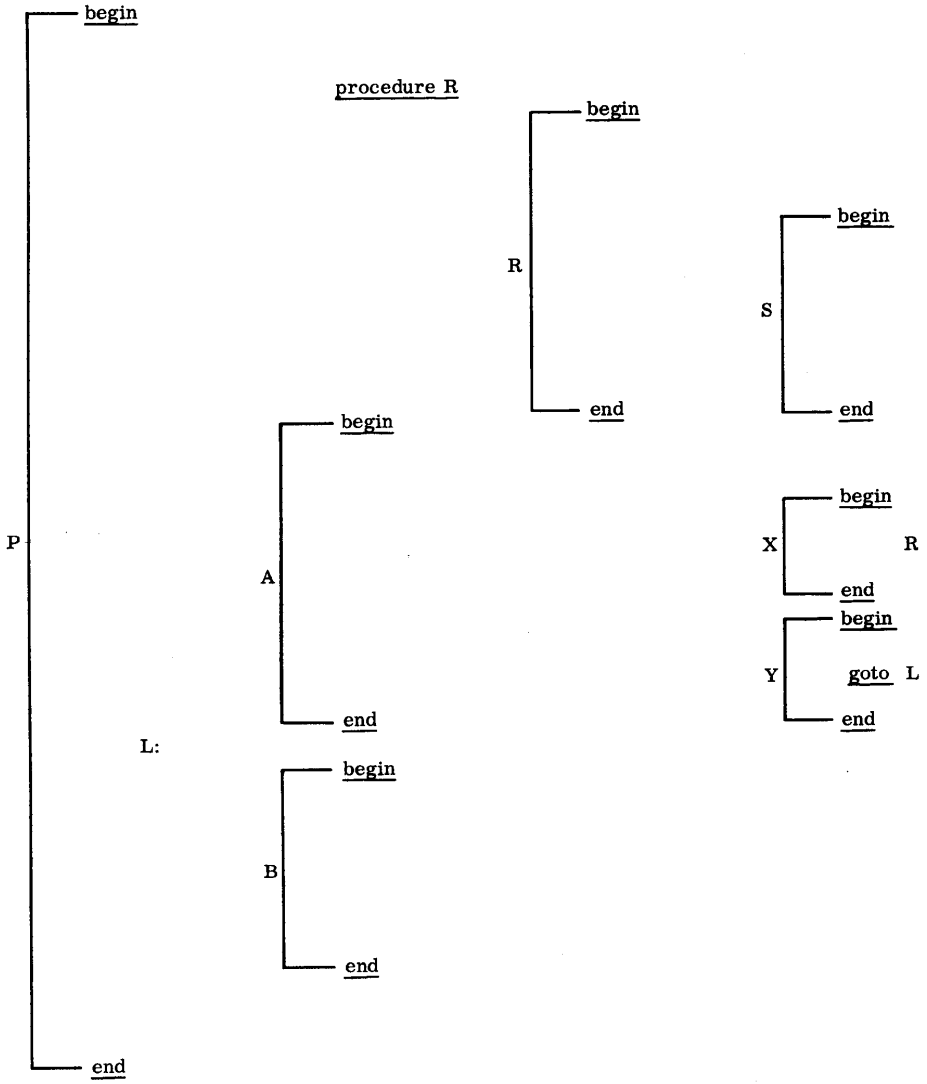
The compiler assigns a block level number to each block in the program, and the object program maintains 32 display entries each of which contains the stack reference for blocks at the corresponding level. The display entry corresponding to the new block is set to contain the new stack reference. Since there are 32 display entries, a program may contain a block structure in which blocks are nested up to a depth of 32 levels.

When a block is exited, the space in the stack occupied by its local variables is released as the variables become inactive. The display entry corresponding to the block being exited necessarily contains the stack reference for this block (the point up to which the stack can be released).

A go to reference from one block in a nest to an outer one results in an exit from that block and from all of the blocks up to but not including the referenced block. Thus, the effect is to change the environment of the active variables to be only those local or global to the referenced block.

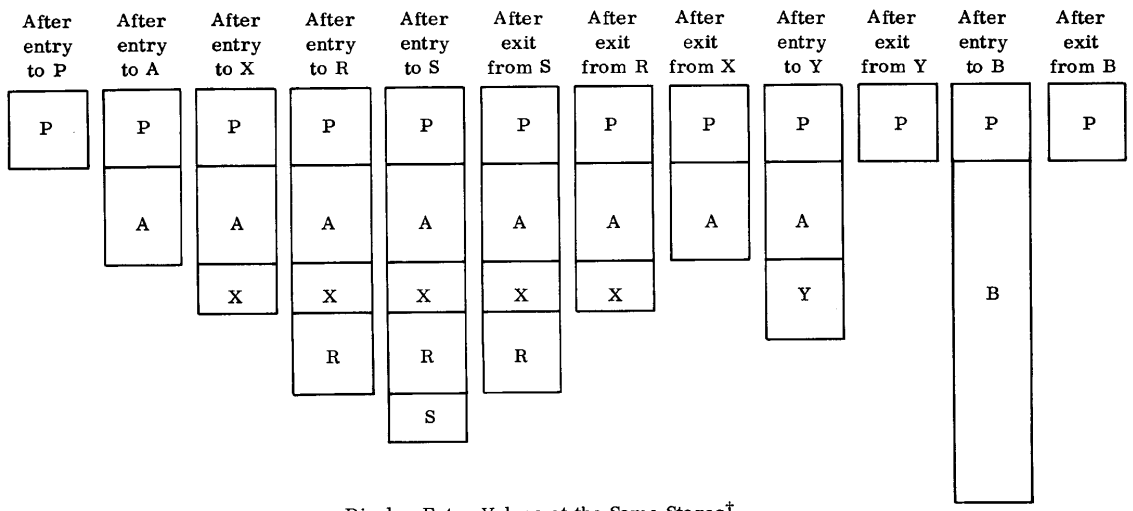
When a procedure call is made, the current environment (or record of it) is preserved, since a return must be made to it at the completion of this call. The environment in which the procedure is declared is established, and the procedure entered. This results in a change of the display, but no stack is released. The procedure is executed with the corresponding variables available to it, and the original environment is re-established.

Consider the following program outline:

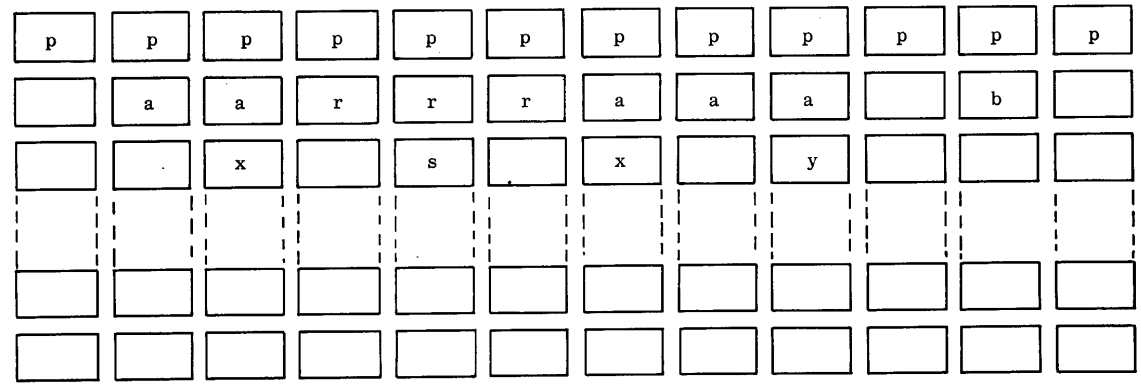


Block P is the program itself; blocks A and B and procedure R are at the same level within P; block S is contained in procedure R; blocks X and Y are at the same level within block A. Block X contains a call to procedure R; block Y contains a jump to label L within the outermost block (the program itself). The changes in the stack and display entries can be visualized as follows:

Stack Contents at Different Stages during Program Execution



Display Entry Values at the Same Stages†



†empty = not used

Following the call to procedure R, the information for blocks A and X remains in the stack; however it is not accessible, since the corresponding display entries are overwritten (and re-established when R is exited). The information for block P also remains in the stack; but this is accessible, since its display entry is not changed. This exactly follows the rules described in the ALGOL-60 Revised Report concerning the accessibility of variables during and after return from a procedure call.

11.1.1 OWN VARIABLES

All own variables are assigned entries in the stack prior to the entries assigned to the outermost block of the program. Thus, own variables are treated as global in definition (local to the whole program), though they are only local in scope to the block in which they are declared, just like other variables.

For an own array, the stack entries for each element in the array appear prior to the entries for the outermost block; the other entries for the array appear in the normal position in the declaration block.

11.1.2 STACK LISTING

The object program controlling system includes a routine which produces the active contents of the stack in a meaningful format upon abnormal object program termination. This structured dump may also be called by the procedure DUMP. Own variables do not appear in such a dump.

11.2 STACK ENTRIES

11.2.1 VALUE OF VARIABLES

Simple local variables and simple formal parameters called by value are represented in the stack as follows:

Real

60-bit entry in standard floating-point form (Section 5.1.3, Chapter 2).

Integer

60-bit entry in standard floating-point form (Section 5.1.3, Chapter 2).

Boolean

60-bit entry in which bits 58-0 are always set to 0. Bit 59 is set to 1 for true and 0 for false.

11.2.2 DESCRIPTION OF VARIABLES

All descriptions of variables in the stack have the following general form:

xxxx	xxxxxx	xxxx	xxxxxx
$\begin{cases} x=0: \langle x\rangle_1 \langle i\rangle_1 \langle o\rangle_3 \langle k\rangle_4 \langle t\rangle_3 \\ x=1: \langle x\rangle_1 \langle \bar{i}\rangle_1 \langle 7\rangle_3 \langle \bar{k}\rangle_4 \langle \bar{t}\rangle_3 \end{cases}$	address 1	$\langle s\rangle_1 \langle o\rangle_{11}$	address 2

x = 0 Transformation required

x = 1 No transformation required

Depending on the value of x, the remaining 11 bits of the upper 12 are either their true values or their one's complement values, as shown above.

t is the type of the variable

t	Type	Possible Use
0	No type	formal and local
1	Integer	
2	Real	
3	Boolean	
4	Integer-real	formal only
5	Integer-real-integer	
6	Real-integer-real	
7	Real-integer	

s is the sign of address 1, wherever applicable

K	Kind	Possible Use
00	Switch	formal and local
01	String	
02	Label of designational expression	
03	No-type procedure	
04	Typed procedure	
05	Array	
06	Constant	formal only
07	Expression	
10	Simple variable	
11	Subscripted variable	

The interpretation of address 1 and address 2 depends on the kind (k) of the description as explained below.

A stack entry representing an arithmetic value may have a structure which makes it appear to be a description.

11.3 DETAILS OF DESCRIPTIONS

The following detailed explanations of the descriptions are ordered according to the kind, k. Return information for a procedure call does not have a kind; it is described first.

11.3.1 TERMINOLOGY

All references to the stack in the object program are relative to the beginning of the stack area for a particular block. When a block is entered at execution time, the base address of the corresponding stack area is assigned. This absolute base address is the stack reference, RRRRRR of the block.

The term segment location, SSSSSS, means an address pointing to a position in the object program. In non-segmented execution, it is the 18-bit complement of an absolute address. In segmented execution, it is interpreted as a 9-bit segment number followed by a 9-bit segment relative address.

The term stack address, AAAAAA, means an absolute address pointing to a particular stack entry.

11.3.2 DESCRIPTION

Return Information

xxxx	xxxxxx	xxxx	xxxxxx
Number of formals	SSSSSS	$\langle s \rangle_1 \langle \text{Appetite} \rangle_{11}$	RRRRRR

s - sign of SSSSSS

Appetite = No. of formals + No. of constants + 1.

SSSSSS Segment location of next sequential instruction following the procedure call.

RRRRRR Stack reference of the block in which the procedure call is made.

00 Switch

xxxx	xxxxxx	xxxx	xxxxxx
5777	AAAAAA	0000	NNNNNN

AAAAAA Stack address of the description of the first element of the switch list.

NNNNNN Number of elements in the switch list.

In a switch declaration, this description is immediately preceded in the stack by the descriptions of the labels or designational expressions (see kind 02, below) which constitute the switch list, as follows:

$\langle$ Designational expression of the nth switch element$\rangle_{60}$

$\langle$ Designational expression of the $(n-1)^{th}$ switch element$\rangle_{60}$

.
.
.

aaaaaa $\langle$Designational expression of the 1st switch element$\rangle_{60}$

$\langle$5777 aaaaaa 0000 n$\rangle_{60}$

01 String

xxxx	xxxxxx	xxxx	xxxxxx
5767	SSSSSS	4000	$\langle c \rangle_1 \langle 0 \rangle_1 \langle X \rangle_6 \langle N \rangle_{10}$

c Format string flag

c = 1 string has been analyzed and can be used as a format string.

c = 0 string must be analyzed to see if it can be used as a format string.

X X replicator count in the string.

N Number of characters in string.

The string itself is stored in-line in the object program.

SSSSSS Segment location of the first word address of the string.

02 Label

xxxx	xxxxxx	xxxx	xxxxxx
5757	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the point in the object program corresponding to the label.

RRRRRR Stack reference of the block containing the label.

For each for statement, the compiler generates an artificial label with the same description as a normal label. This label is used to return from the end of the for statement to the control at the beginning. Whenever the for statement is not in execution, the segment location, SSSSSS, of this label is set to point to a special system entry segment location 000011, in order to detect abnormal use of the statement. In addition, bit 59 of the description is preset to 1 before entry to each step-until element, and set to 0 after this element has been entered.

02 Designational Expression

xxxx	xxxxxx	xxxx	xxxxxx
5757	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the code which evaluates the expression and jumps to the resulting address.

RRRRRR Stack reference of the block containing this code.

03 No-type Procedure

xxxx	xxxxxx	xxxx	xxxxxx
7747	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the procedure.

RRRRRR Stack reference of the block containing the procedure.

04 Typed Procedure

xxxx	xxxxxx	xxxx	xxxxxx
{004t / 773t̄}	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the procedure.

RRRRRR Stack reference of the block containing the procedure.

05 Array

xxxx	xxxxxx	xxxx	xxxxxx
{205t / 572t}	AAAAAA	0000	DDDDDD

AAAAAA Base address of the array elements in the stack.

DDDDDD Base address of the dope vector in the stack. The dope vector is used to calculate the addresses of the array elements (see below).

The elements of an array are assigned above the last working location of the particular block.

own arrays are handled in the same way, except that their elements are assigned among the own variables.

The elements of an array called by value are copied (and transformed, if necessary) to a position above the working locations of the block of the procedure.

In an array declaration, the dope vector of the corresponding bound-pair list precedes the descriptions for all array identifiers of an array segment.

The dope vector for the array declaration

array A $[\ell_1 : u_1, \ell_2 : u_2, \ldots \ell_n : u_n]$ is:

	<		$C_n > 60$
	<		$C_{n-1} * C_n > 60$
			.....
			.....
	$< C_2 * C_3 * \ldots\ldots\ldots\ldots$		$* C_{n-1} * C_n > 60$
	<	Length of array	> 60
DDDDDD	<	Lower bound effect	> 60
	<	n = No. of dimensions	> 60

where $C_i = u_i - \ell_i + 1$

Length of array = $C_1 * C_2 * C_3 \ldots\ldots * C_n$

Lower bound effect = $(((\ldots(\ell_1 * C_2 + \ell_2) * C_3 + \ell_3) * \ldots) * C_n + \ell_n$

The address of any element is referenced by the base address of the array plus $(((\ldots(i_1 * C_2 + i_2) * C_3 + i_3) * \ldots.) * C_n + i_n$ - lower bound effect.

For example, the description of the declaration

array A, B [1:3, 2:5] is:

```
                <                                          4>60
                <                                         12>60
ddddddd         <                                          6>60
                <                                          2>60
                <5725        bbbbbb        0000        dddddd>60
                <5725        aaaaaa        0000        dddddd>60
```

06 Constant

xxxx	xxxxxx	xxxx	xxxxxx
{206t / 571t}	000000	0000	AAAAAA

AAAAAA Address of the constant in the stack.

07 Expression

xxxx	xxxxxx	xxxx	xxxxxx
{007t / 770t}	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the code which evaluates the expression.

RRRRRR Stack reference of the block containing this code.

10 Simple Variable

xxxx	xxxxxx	xxxx	xxxxxx
{210t / 567t}	000000	0000	AAAAAA

AAAAAA Address of the simple variable in the stack.

11 Subscripted Variable

xxxx	xxxxxx	xxxx	xxxxxx
{011t / 766t}	SSSSSS	4000	RRRRRR

SSSSSS Segment location of the code to evaluate the address of the subscript variable.

RRRRRR Stack reference of the block containing this code.

OBJECT-TIME ABNORMAL TERMINATION DUMP 12

Upon detection of fatal error during program execution the ALGOL control routines perform the following actions:

1. Empty all output format areas onto associated files.
2. Print diagnostic on standard output file, channel 61.
3. Print a structured dump.

Chapter 8 describes object-time diagnostics.

The amount of information given by the dump may be changed, from its default, by the execution time option parameter E (chapter 13).

E = O	Error message only
E = T	Error message and traceback
E = V	Error message, traceback, local and formal variables
E = W	Error message, traceback, local and formal variables

The arrays may be dumped and decimal translation obtained if either of the last two options is selected.

12.1 STRUCTURED DUMP

The structured dump traces back the execution path from the point where the error occurred, through the block structure, to the entry point of the program. Information relevant to the program will be selected from store in accordance to the E option. The dump has the following format:

```
THIS ERROR OCCURRED AFTER LINE      XXXX
IN THE BLOCK ENTERED AT LINE        XXXX    IN YYYYYY

(global information)
(environmental information)

THIS BLOCK WAS CALLED FROM LINE     XXXX
IN THE BLOCK ENTERED AT LINE        XXXX    IN YYYYYY

(environmental information)
```

```
THIS BLOCK WAS CALLED FROM LINE     XXXX
IN THE BLOCK ENTERED AT LINE        XXXX    IN  YYYYYY

(environmental information)


(environmental information)

.............

.............
```

LINE XXXX refers to the line number counted, from zero for the first card, in the program or code procedure deck. It is printed for every tenth line on the source listing (L-option on the ALGOL card control, Chapter 6).

If the block entered is a standard procedure (3.2.4) then.... appears instead of the line number.

YYYYYY refers to the deck in which this block was declared. It will be either the program name default XXALGOL, the code procedure identifier (e.g. CP00001 for 'code' 1), or if the block is a standard procedure, it will be the procedure name.

12.2 GLOBAL AND ENVIRONMENTAL INFORMATION

Each item of the global and environmental field consists of an address, A, printed as 6 octal digits and the contents C of that address printed as 20 octal digits in fields of 4,6,4, and 6. The layout is as follows:

Address Field	Information Field
XXXXXX	XXXX XXXXXX XXXX XXXXXX

When decimal translation is required, the contents, C, are checked. A floating point number is printed in standard format (Chapter 3.1, Section A.5), otherwise it appears in octal. Occasionally a floating point number with a very small negative or large positive exponent will not be translated. When converted the layout is as follows:

Address Field	Information Field
XXXXXX	±X.XXXXXXXXXXXXX'±XXX

For environmental information, the address A also is printed as an increment on the stack reference, making the layout as follows:

Stack Reference Index	Address Field	Information Field
XXXXXX	XXXXXX	XXXX XXXXXX XXXX XXXXXX

12.2.1 GLOBAL INFORMATION

The global information applies to the running program as a whole, without regard to the currently active block structure. It has the following format:

```
THE GLOBAL VARIABLES ARE:
UA, VALUE  )
UV         }    information items.
LASTUSED   )
```

UA, UV and LASTUSED name variables internal to the ALGOL system.

The information item for UA is the address and value of the last accessed of the following:

a format parameter

a typed procedure

an array element

UV is used only to contain either the value of the last accessed formal parameter (if it does not appear in the stack) or the value of a typed procedure. The address of UV is always zero.

LASTUSED contains the address and value of the top stack element. It is the current top stack entry and will not necessarily be the highest entry so far, as some blocks may have been exited.

12.2.2 ENVIRONMENTAL INFORMATION

Environmental information consists of descriptors or values of formal and/or local variables belonging to the block currently being dumped. Arrays being dumped appear after the local variables. Formal variables appear only if the particular block is a procedure.

Simple local variables and simple formal parameters called by value are represented by their values; all other variables are represented by a descriptor. The values are always stored in floating point; the descriptors are as described in Chapter 11.

On the extreme left-hand side of the dump of formals, locals, and arrays, appears the index on the stack reference for the first variable on that line. This index is a negative increment on the current stack reference for formal parameters and a positive one for local and array variables. It is the same as REF on the identifier table obtained under compilation with the M-option (Chapter 6).

Formal Variables

Formal variables are dumped in the following structure.

```
THE FORMAL VARIABLES ARE..

1st line      :        last constant used as actual parameter
   :          :        :
   :          :        :
              n+1      1st constant used as actual parameter
              n        last formal parameter
              :        :
              :        :
          000001       1st formal parameter
          000000       Return information
```

Local Variables

Local variables are dumped up to 4 items per line. If the contents of one is repeated in consecutive variables, the first one is printed, followed by:

```
.UNTIL.     XXXXXX     ZZZZZZ
```

where XXXXXX is the stack reference index and ZZZZZZ is the address of the first item with a different value. Entries in the stack are stored in the reverse order to their declaration in the source program, and they are dumped in this order. One stack entry is generated for each of the following:

Every declared variable, whether real, integer, Boolean, procedure, label, array, etc.

Each for statement generates an artificial label.

Each designational expression in a switch list generates a dope vector of length n+2.

The working storage includes the local variables of standard procedures, partial results in the computation of long expressions, and any space allotted by the compiler for optimization. If requested by the presence of the W option, the working storage is dumped as a continuation of the local variables.

If requested by the presence of the A option, the arrays are dumped after the local variables for that block. The layout is as follows:

```
THE ARRAY ELEMENTS ARE..
DESCR... XXXXXX
(elements of 1st array)
DESCR... XXXXXX
```

(elements of 2nd array)

.
.
.

XXXXXX is the index on the stack reference for the descriptor of array that follows. The name of a particular array may be found by reference to XXXXXX in the dump of the local variables or in the cross reference map produced by the M option on the ALGOL card. Elements of arrays are dumped in the order of storage (Chapter 11.3). The stack reference index for the elements of an array is the index on array element zero; e.g. A [0,0].

EXECUTION TIME OPTIONS 13

Execution time options, which control the running of an ALGOL-60 object program, may be specified either by control cards or by directives on the standard input device. The latter method is provided so that programs executing in segmented mode can select run time options.

(Nomenclature: <name> means that name is a description of the indicated field rather than the literal contents thereof. The two characters < > are used only in this way and do not appear in any fields. {<contents>} means that contents may be supplied or omitted. The separator | between fields means that a choice must be made between the alternative formats thus separated.)

Control card options are provided as parameters on the control card (such as program call card) which initiates execution of the ALGOL program, for example:

LGO (<options>)

The run time directives are supplied, along with any CHANNEL cards, as the first data on the standard input device. Such directives must appear before the first data card, an OPTIONS, END card, or a CHANNEL, END card. The card format is:

OPTIONS, <options>

The eight characters, OPTIONS, appear in columns 1-8. OPTIONS, END may be employed to terminate recognition of OPTIONS cards; it may be omitted.

Since it is functionally identical to a CHANNEL, END card, it should not be used if CHANNEL definitions remain to be processed. Conversely, a CHANNEL, END card will also suffice to terminate input of OPTIONS cards, if required.

If both program call card and OPTIONS card directives appear,the OPTIONS card definitions take precedence.

Each installation will have its own default values for these parameters.

The format of the options field follows:

<options> may be option-i	<option-1> { ,..., <option-k>} is S {0=} or D {=0 | =n} or C {=n} or E {=0 | = <character string>}

13.1 STACK STATISTICS (S)

The S option takes the form S=O or S. The former selects no production of statistics. S alone will result in the monitoring of the size of the stack during program execution, and upon abnormal termination, the maximum address reached by the stack will be recorded on the standard output device. Absence of the S option will cause the installation default to hold. Default of S=O is recommended since monitoring will degrade performance in a program with large block activity.

13.2 DEBUGGING OUTPUT (D AND C)

These options are relevant only when the program has been compiled in D mode (debugging) to activate debugging directives in the source code.

D option form:

D=O	
D	
D=n	n is a positive integer

C option form:

C	
C=n	n is a positive integer

D=O suppresses any debugging output resulting from D mode compilation. These outputs however, will be computed in the object code; otherwise the debugging output will appear on the channel specified by C, C=n or by default.

D overrides a possible default option of D=O. It will cause all debugging output to appear with no limit on the number of messages.

D=n is similar to D except that output will be suppressed after the number of debugging messages specified by n have appeared. Program execution will continue.

C selects the standard output device as the recipient for debugging outputs.

C=n selects channel specified by n as the recipient for debugging outputs. This channel must be defined by channel card, unless n is 61; channel 61 is defined automatically for all ALGOL runs.

13.3 ABNORMAL TERMINATION DUMP FORMATS (E)

E option forms:

E = O
E
E = < character string>

When E = O, upon abnormal termination of execution, the abnormal dump is truncated; it provides only the error message and the point of error.

E overrides a possible default of E = O, and results in a dump of the active stack with scalars and descriptors. It is equal to E = V as defined below.

The other E options provide varying degrees of detail in the abnormal termination dump according to the combination of the following character codes to form <character string.>

T	include traceback (flow from block-to-block)
V	include local and formal variables
W	include working storage (temporaries)
A	include arrays
X	provide decimal translation (X) of stack values

The codes T, V, W are progressively more inclusive each being implied by the following one. Since A refers to a particular form of local variables and X refers to translation of the contents of stack locations they are meaningful only in combination with V and W. Thus, the set of all options is:

T, V, W, VX, WX, VA, WA, VAX, WAX.

13.4 EXAMPLES

```
LGO (E=VX,S,D=O)
```

This control card initiates execution with no debugging output. Upon normal termination, the highest stack address is output. On abnormal termination, the dump is a translation of active stack with locals and formals; but arrays and working storage are not dumped.

```
OPTIONS, E=VX, D=O
CHANNEL, 5=60
OPTIONS, S
OPTIONS, END
```

These directives (intervening channel card shown) have identical effect to the preceding example. This method of specifying execution time options is convenient if a program is executing in segmented mode or from a terminal in interactive mode.

```
LGO (E=WA, D=2000, C)
```

The object program will produce debugging outputs on channel 61 normally until 2000 calls to the debugging procedure have taken place. Thereafter, the debugging procedure will produce no output. In event of abnormal termination, a complete stack dump will ensue, including array elements and working storage areas. This dump will be in octal.

```
OPTIONS,E=WA,D=2000,C=4
CHANNEL,4=61
CHANNEL,END
```

The effect of these directives is identical to that of the preceding example.

DEBUGGING FACILITIES 14

A source program may contain debugging directives designed to detect faults during subsequent program execution. Under the normal mode of compilation, such directives are ignored by the compiler and thus have no effect during execution of the resultant object code. However, if the debugging option (compile time D option) is selected, any debugging directives present are detected by the compiler and cause debugging code to be inserted into the object program. Since these debugging directives are acceptable ALGOL-60 character sequences, it is unnecessary to remove them from a source program submitted to an ALGOL compiler in which the directives are not recognizable.

14.1 GENERAL

The debugging directives, trace and snap are keyed to identifiers in the source program. Trace monitors the flow of control through a program; snap monitors changes in the values of program variables. The trace directives apply to identifiers which represent labels and procedures, and the snap directives apply to identifiers which represent simple variables or arrays. An identifier which is the object of a snap directive may be monitored in certain specified areas of the source program by judicious placing of the snap directive and by use of the off directive to return the identifier to normal status. However, a trace directive causes an associated identifier to retain that attribute permanently, so that it is monitored throughout program execution.

The debugging directives affect the object code only when the debugging option has been specified on the compiler control card. The object program will include code which performs monitoring during subsequent execution by producing messages either on the standard output file or on any other channel selected, at execution time, to receive debugging messages (C option). If an object program has been compiled under the debugging option, debugging output can be suppressed or limited by the D option. The source program should be recompiled in non-debugging mode to obtain an efficient object program.

14.2 DEBUGGING DIRECTIVES

Debugging directives may be present in the source program at all stages of its development whether or not they are to be activated. This is achieved by requiring debugging directives to be included only within commentary sequences. By this means, it is possible to maintain debugging directives in a source program which is compatible with several other compilers.

A debugging directive takes the form of <debugging directive> in the following definition:

```
    <identifier list>  : : = <identifier> | <identifier>, <identifier list>
        <directive>    : : =  trace | snap | off
<debugging directive>  : : = <directive> <identifier list>
```

e.g. snap a,b,marilyn
trace L1
off b

(trace, snap, off have the same hardware representation as compound delimiters)

The debugging directives are inserted in a source program within commentary sequences according to the following rules. A commentary sequence may contain only one debugging directive. Any commentary sequence may be expanded to contain a debugging directive by inserting it after the last symbol in the sequence immediately before the semicolon which terminates the sequence. For example:

```
; comment *** snap a,b;
begin comment first of all trace L1, Failure;
; comment off a ;
```

(Directives will not be recognized in text following end.)

In the absence of the debugging option at compile time, the previous examples would be equivalent to semicolon, begin, semicolon respectively. However, in debugging mode (D option), all comment structures are examined to detect the presence of trace, snap, or off. A syntax error will result if the remainder of the commentary sequence does not conform to the above definition. Erroneously constructed debugging directives will not be recognized as such in the absence of the D option. The following sections assume that a syntactically correct debugging directive of the appropriate type has been encountered and that the debugging option is enabled.

14.3 TRACE DIRECTIVE

This directive monitors the flow of control within an object program.

```
trace    id-1,....,.,id-n
```

The identifier list must contain at least one identifier; order within the list is irrelevant, and any member of the list will be referred to as a trace-identifier. A trace directive may appear anywhere in a block, but the identifiers in the list each must be declared in the same block as labels, typed procedures, or no-type procedures. The trace directive does not refer to any previous declaration of a trace identifier in an outer block or in a disjoint block. Nor can the trace directive refer to some subsequent declaration of a trace identifier in an inner block (this will cause a normal re-declaration of the identifier).

Any transfer or reference to or call of a trace-identifier while its associated declaration is within scope, or by means of an actual-formal correspondence, will be monitored throughout program execution. It is not possible to suppress monitoring within specific areas of the source program.

The format of the messages produced by trace monitoring at execution time is described in the following sections.

14.4 SNAP DIRECTIVE

This directive monitors changes in value of specified program variables during object program execution.

```
snap id-1,.....,id-n
```

The identifier list contains at least one identifier. The order within the list is irrelevant, and any member of the list will be referred to as a snap-identifier. The snap directive does not require the associated identifiers to be declared in the same block. The snap directive can appear anywhere in any block depending on the specific areas of the source program in which monitoring of snap-identifiers is required.

A snap-identifier must be declared as a simple variable or as an array. It may be declared in a block contained in that in which the snap directive occurred or in a block which embraces the block in which the snap directive is to occur.

In the former case, the snap-identifier is monitored throughout the scope of its declaration (until re-declared or until the declaration block is left). In the latter case, the snap-identifier is monitored only in statements between the occurrence of the snap-directive and the end of the block in which the snap-directive occurred, including statements in inner blocks.

Since the snap-directive is not itself executable, the interpretation being made at compile-time only, the above definitions of the range of a snap-directive are static (lexicographic) and refer to the physical program sequence rather than to the sequence in which the program is executed.

The statements which cause monitoring of a snap-identifier to take place at execution time are dependent upon the kind of variable represented by the identifier; therefore a list of statements and corresponding message formats is given in the following sections.

14.5 OFF DIRECTIVE

This directive provides additional control over the monitoring process initiated by a snap directive.

```
off    id-1,.....,  id-n
```

The identifier list contains at least one identifier; order within the list is irrelevant. If any identifier in the list is a snap-identifier at the point at which the off directive is encountered, monitoring of that identifier will not occur in succeeding statements unless the identifier appears in a

subsequent snap directive. The off directive is thus the logical opposite of the snap directive; since it can appear in any part of the source program, it provides statement-by-statement control of the monitoring of snap-identifiers. Upon exit from the block in which a snap directive occurred, the effect of the directive terminates just as if a corresponding off directive had appeared at the end of the block. The off directive cannot be used to terminate monitoring of trace-identifiers.

The above definitions of monitoring ranges are physical and not logical, since the debugging directives are interpreted at compile time only. Thus, if an off directive is specified in a conditionally executed area of the source program, no monitoring code will be assembled for subsequent references to the associated identifiers irrespective of the actual flow of control at execution time.

A snap-identifier will be subject to monitoring in the statement subsequent to:

the occurrence of the snap directive, or

the first declaration of the identifier,

whichever occurs the later; and in all lexicographically subsequent statements until the occurrence of an off directive for that identifier, or exit from the block in which the snap directive occurred, or entry to a block in which the identifier is redeclared, whichever occurs the earliest. If monitoring is suspended by the last named event above, it will be resumed upon exit from the block in which the redeclaration occurred.

If a snap-identifier occurs in a snap directive in a block in which the snap-identifier is undeclared, monitoring will occur for any subsequent disjoint declarations (of suitable kind) in inner blocks.

To monitor usage of an identifier not only in a certain block but within a procedure declared in that block which may modify the identifier, it is necessary to insert the snap directive ahead of the procedure body (among declarations in that block).

Examples:

```
begin
    integer i ;
    comment  snap  i ;
    procedure  p(x); integer x ;
    begin i :=i+x; comment i is monitored;

if i = j then
    begin comment snap i ;
        i : = complicated (x, y, z);
            comment off i; j : = i + 1 end else
```

14.6 DEBUGGING OUTPUT

The format and content of debugging output depends both upon the kind of variable being monitored and upon the context. Information that may be obtained through debug directives is described in the following sections categorized according to the kind of the monitored variable; a summary of the general formats is given in this section.

A single item of debugging output is referred to as a monitoring message. In the following list of monitoring messages the portions in lower case are replaced during execution by the line number, identifier, or value. The identifier produced will correspond to that used in the debugging directive; but if the original identifier contained more than ten characters, only the first nine will appear in the monitoring message.

** LINE linenumber (GO TO) identifier
... produced for labels trace

** LINE linenumber (INVOKE) identifier
... produced for procedures, functions trace

** LINE linenumber (USAGE) identifier
... produced for arrays, simple variables, subscripted variables snap

** LINE linenumber identifiers: = value
... produced for simple variables snap

** LINE linenumber (EXIT) identifier = value
... produced for simple variables snap

** LINE linenumber identifier [*] : = value
... produced for subscripted variables snap

Examples appear in the following sections.

14.7 LABEL MONITORING

When applied to a label identifier, the trace directive produces monitoring messages which indicate when and from where a transfer to that label occurred. A trace directive referring to the label identifier must be inserted at some point in the block in which the label is declared (just before the label, or in the block head, for example).

The flow of program control may pass through a label either by a go to statement or sequentially after completion of a preceding non-transfer statement. No monitoring message will be produced for sequential execution, but all transfers of control by go to statements will be monitored. Included are cases where the designated label is obtained as a switch element, as the value of a designational expression, or by use of a corresponding formal parameter label identifier.

The monitoring message produced during execution contains the line number from which the transfer occurred:

```
**LINE 231   (GO TO)   INFEAS
```

A transfer to the label, INFEAS, occurred at line number 231 in the source program. This is not the line number at which INFEAS occurs, that line number is given in the identifier map produced by the M option.

14.8 PROCEDURE MONITORING

When applied to a procedure identifier, the trace directive produces monitoring messages during execution which indicate when and from where that procedure has been invoked. A trace directive must be inserted at some point in the block in which the procedure is declared (prior to the procedure declaration in the block head, for example).

Procedures may be invoked either by a procedure statement (no type procedure) or by the appearance of a function designator (typed procedure) in an arithmetic or Boolean expression. Monitoring will be produced unless it is a code-bodied procedure or represents a standard procedure or standard function. Use of this latter class of procedures as trace identifiers is not permitted.

The monitoring message produced during execution contains the line number from which the procedure was invoked. The following example:

```
** LINE 235 (INVOKE) PRODUCT
```

Procedure, PRODUCT, was invoked from a statement on line number 235.

14.9 SIMPLE VARIABLE MONITORING

The snap directive permits monitoring of arrays and simple variables. For simple variables, the snap directive monitors changes, or possible changes, in the value of integer, real or Boolean variables.

To monitor a variable, the associated identifier must be specified as a snap-identifier according to the rules given in section 14.4. Monitoring can be terminated on a statement-by-statement basis through the off directive.

The value of a simple variable may be changed if it is:

one of the left parts in an assignment statement

the controlled variable of a for clause

used as a call-by-name actual parameter corresponding to a formal which undergoes a change of value

The monitoring message for a direct assignment to a monitored variable takes the following format:

```
** LINE 71   INDEX : =9
```

This message contains the new value of the variable in addition to the line number and the identifier, INDEX. The type of the monitored variable is integer. If the variable type is real, the value is output in decimal standard format. If the variable type is Boolean, the output representation is 'TRUE' or 'FALSE'.

Monitoring of changes in value caused by actual/formal correspondence is more complicated. If the variable appears in the actual parameter list of a procedure statement or a function designator and is not specified as call-by-value, its value will change whenever the corresponding formal variable is altered. Since the monitoring process should indicate all possible changes in the value of a snap-identifier variable, and since the bodies of procedures and function designators are not necessarily available at compilation time, it is assumed for debugging purposes that a variable may be altered whenever it appears by itself as an actual parameter. All such usages of the snap identifier will result in a monitoring message at execution time.

The format of the messages produced depends upon the context of the usage of the snap-identifier as a parameter, since it is not always certain that control will be returned to the code that will monitor the final value of the variable. Thus, the following example of a monitored variable as an actual parameter in a procedure statement may produce one or two messages depending upon the behavior of the procedure:

```
comment check the determinant, snap DET;
INVERT (array, DET);
comment DET is the determinant of array, off DET;
```

The following type of message always will be produced:

```
** LINE 102 (USAGE) DET
```

A second type of monitoring message may be output also:

```
** LINE 102 (EXIT) DET = 17294
```

However, if the procedure INVERT does not return control directly to the calling block, only the first message is produced. Although the monitoring process cannot always provide the (EXIT) message giving the final value of the monitored variable, the value of the variable cannot change without being indicated by the appearance of a (USAGE) message.

The use of a monitored variable as an actual parameter of a function designator poses similar problems, but it is assumed here - for debugging purposes - that a function designator will always return control to the expression or statement from which it is invoked. Therefore, it is possible to record the value of the variable upon exit from the function; and upon termination of the invoking statement, such as an assignment statement, an (EXIT) type monitoring message is produced. For example, the source code:

```
comment snap M, N;
    M := 2* unstack (N) + 1;
```

would produce the following messages

```
** LINE 115 M := 11
** LINE 115 (EXIT) N = 5
```

However, under the following circumstances, the exit value cannot be directly monitored. In such cases, the function designator occurs in a designational expression, in a then or else expression of a conditional expression, or in the if clause of a conditional statement. In these circumstances the (USAGE) type of message indicates the possibility of a change in the value of the monitored variable. For example the following source code:

```
comment snap N;
go to if j = 0 then L1 else switch [unstack (N)];
```

would produce the message

```
** LINE 116 (USAGE) N
```

14.10 ARRAY AND SUBSCRIPTED VARIABLE MONITORING

The snap directive applied to array identifiers is used to monitor occasions on which the elements of the array may have been altered. Since the elements are dependent on the execution time value of one or more subscript expressions, monitoring supplies only the point of usage.

One exception is direct assignment to the array element as a result of its appearance as a left part in an assignment statement. The following source text:

```
comment snap A;
A [j, 3 * j - 2] := 2 ** 5;
```

would produce the following message:

```
** LINE 127 A[*] := 32
```

The monitoring message contains the new value of the subscripted variable, integer, real, or Boolean but does not supply the values of the subscript expressions.

Apart from direct assignment, the values of an array may change either if it is the controlled variable of a for clause or if it or one of its elements is used as a call-by-name parameter of kind array or subscripted variables. Use of an array or subscripted variable is considered to be capable of changing the array values if the actual parameter consists of the array identifier or a single subscripted variable. Thus the following source text:

```
comment snap A, B, C;

if i = 0 then p (A, B [j] + 1, C [j] );
```

where A, B, C are array identifiers, would cause the following type of message to appear if i were zero:

```
** LINE 53 (USAGE) A
** LINE 53 (USAGE) C
```

Use of a subscripted variable or an array as an actual parameter of a function designator is treated as in the case of simple variable parameters. Normally, such usage would result in the following type of message on termination of the invoking statement:

```
** LINE 62 (EXIT) B
```

indicating that elements of the monitored array, B, may have new values on exit from the body of the function designator. Use of arrays or subscripted variables as parameters of function designators in the if clause of a conditional statement, in the "then" or "else" expression of a conditional expression, or in a designational expression will produce the (USAGE) message; for example:

```
i : = if j = 0 then manip ( A ) else j;
```

would produce a message of the type;

```
** LINE 85 (USAGE) A
```

showing that the typed procedure, manip, could have altered any of the elements of the array, A.

Finally, a subscripted variable as the controlled variable as the controlled variable of a for clause is also monitored by a USAGE message if the array identifier is a snap-identifier at that point in the program. Thus each assignment to the controlled variable, A [i, j] for instance, would be accompanied by a message of the type:

```
** LINE 87 (USAGE) A
```

No attempt is made to record the values of an array in the monitoring messages, as the elements altered are not easily established at compilation time. The monitoring process is primarily concerned with identifying critical usages of array identifiers rather than the detailed results of those usages. The only exception is the direct assignment to an element in which case the new value is provided.

If preliminary use of the snap directive on an array identifier is not sufficiently detailed or too indiscriminate, a more selective monitoring should be attempted by using snap directive on a temporary variable introduced to sample critical array elements at the points indicated by the original monitoring of the array itself.

ALGOL MACROS 15

A number of COMPASS coded macros are provided to expand the areas of application of ALGOL-60 programs. This may be achieved either by enabling ALGOL programs to make use of subprograms coded in a different language or by providing special purpose COMPASS coded procedures to permit ALGOL programs to perform tasks which require the compactness of extreme code efficiency.

Macros which serve the former purpose of providing linkage to subprograms coded in another language will be referred to as interface macros. Macros which facilitate the production of special purpose COMPASS procedures will be referred to as utility macros. Utility macros are further subdivided into entry-exit macros, specification macros and formal-handling macros.

15.1 INTERFACE MACROS

Each macro provides a closed COMPASS subroutine that may be called directly from an ALGOL main program. The expansion invokes an existing external procedure (written in a specified language), provides the actual parameters given in the originating ALGOL procedure statement or function designator, and returns control to the appropriate point in the ALGOL main program.

ALGOL actual parameters will be checked to ensure they are meaningful to the language in which the external procedure is written; but consistency of internal data representation is not checked (e.g., an ALGOL integer array stores its elements "row-wise" and each cell contains a normalized floating point integer). Certain operations in the external procedure, notably input/output, may be possible only if the main program was of the same language, which generally will not be the case.

The macro can be used as the only statement in the body of a COMPASS interface deck, which after assembly, must be loaded with the ALGOL main program and the external procedure. If more than one external procedure is to be linked to the ALGOL program, however, several macros should be combined into one interface deck.

15.1.1 FTNLINK

The following macro interfaces a FORTRAN Extended subroutine or function to an ALGOL main program. The ALGOL program must invoke a code declared procedure to execute the FORTRAN Extended subprogram. The name of this procedure is arbitrary; only its code number is required by the interface macro.

```
        name    FTNLINK    number
```

name: {location field} entry point of FORTRAN Extended subroutine or function.

number: {up to 5 digits} the code number used in the declaration of the associated procedure in the ALGOL main program.

ALGOL parameters switch, label, string, or procedure will result in the diagnostic PARAMETER KIND ERROR. All other parameter kinds will be passed to the subprogram in a suitable form for FORTRAN processing.

Example:

ALGOL usage:	ftnsub (x, y, z);
ALGOL declaration:	procedure ftnsub (a, b, c); code 73;
Interface macro:	PGM1 FTNLINK 73
FORTRAN definition:	SUBROUTINE PGM1 (P, Q, R)

15.1.2 RUNLINK

The following macro interfaces a RUN subroutine or function to an ALGOL main program. The ALGOL program must invoke a code declared procedure to execute the RUN subprogram. The name of this procedure is arbitrary; only its code number is required by the interface macro.

```
name RUNLINK number, nopars
```

name: {location field} entry point of RUN subroutine or function.

number: {up to 5 digits} the code number used in the declaration of the associated procedure in the ALGOL main program.

nopars: {absolute address expression} the number of formal parameters in the RUN subprogram.

ALGOL parameters switch, label, string, or procedure will result in the diagnostic PARAMETER KIND ERROR. If the number of actual parameters differ from the value of nopars the diagnostic, PARAMETER COUNT ERROR will be produced.

Example:

ALGOL usage:	t : = arcsine (x) ;
ALGOL declaration:	real procedure arcsine (z) ; code 153;
interface macro:	ASIN RUNLINK 153, 1

15.2 ENTRY/EXIT MACROS

The following macros are intended for use within special purpose COMPASS subprograms. They provide entry from a procedure statement or function designator in an ALGOL main program, and return, with a value in the latter case, to the appropriate point in the invoking program.

15.2.1 CODE

This macro provides linkage between the code declaration of an external procedure in an ALGOL main program and the corresponding COMPASS subprogram. Control will transfer to the CODE macro expansion and the statements which immediately follow the macro will constitute the first executable code in the procedure. The macro expansion creates a new block level in the stack and establishes the environment for the formal handling and specification macros to be described in the following sections.

```
name    CODE    number
```

name: {optional location field parameter, up to 7 characters} an arbitrary name for the COMPASS subprogram initiated at this point. This name will be used during traceback in event of an execution time error. If omitted, it is assumed to be CP→number.

number: {up to 5 digits} the code number used in the declaration of the associated procedure in the ALGOL main program.

Example:

```
ALGOL usage:          i := extract (p, n) ;
ALGOL declaration:    integer procedure extract (position, word); code 52;
entry macro:          UNPACK CODE 52.
```

Since this macro name duplicates the name of a COMPASS pseudo operation, the latter has been renamed CODE. so that it is still available whenever the ALGOL text is used.

15.2.2 RETURN

This macro causes a normal return from a COMPASS coded procedure. Although the procedure may terminate in other ways (e.g., GOTO macro) this will be the normal point of exit, and will return control to the point in the main program from which the CODE macro was entered. If invocation was by a function designator, the value of the function must be present in an X register at this stage.

```
RETURN      xreg
```

xreg: (optional X register designator) x register contains the floating point value of the function designator represented by the preceding code. If xreg is omitted, it is assumed the code was executed from a procedure statement.

Examples:

```
RETURN              . from procedure statement
RETURN    X4        . within expression
```

15.3 SPECIFICATION MACROS

The following macros are used within a COMPASS coded procedure to assist in establishing the number and kinds of actual parameters in the current invocation of the procedure. Contents of B4 must always be saved and restored before calling any macro.

15.3.1 PARAMS

The following macro provides the number of actual parameters in the current invocation of the COMPASS procedures. No check is performed on the parameter count.

```
PARAMS    Xreg
```

Xreg: (X-register designator) X register will receive the number (in binary fixed point) of actual parameters transmitted in the current call.

Example:

```
PARAMS    X5
```

15.3.2 KIND

This macro provides a numerical representation of the kind of an actual parameter. To relax the normal restriction that ALGOL actual parameters are checked against a fixed specification, this macro does not produce any execution-time diagnostics. No check is made that the parameter has been supplied in the current invocation.

```
KIND    param, xreg
```

param: (absolute address expression, or X register designator) the value or contents of param is the ordinal of the actual parameter to be examined, e.g., 2 indicates the second parameter.

xreg: (X register designator) designated X register will receive numerical kind code of the specified actual parameter. The codes corresponding to each kind of actual parameter are:

switch	0
string	1
label	2
no type procedure	3
typed procedure	4
array	5
constant	6
expression	7
simple variable	8
subscripted variable	9

examples:

```
KIND    3,X1
KIND    FORMAT, X6
KIND    X2, X3
```

15.3.3 SPEC

This macro permits the programmer to restrict the kinds of an actual parameter. Since allowable kinds for a given parameter may be run-time dependent, this check occurs in-line; it is assembled at the location of the macro call rather than in centralized specification checking code. This macro may produce abnormal termination with a diagnostic, and does not provide direct information about the kind of the actual parameter:

```
SPEC    param, spcodes
```

param: (absolute address expression, or X register designator) the value, or contents, of param is the ordinal of the actual parameter to be examined.

spcodes: (string of up to 10 alphabetic characters) a string of one or more single character spec codes, where each is an alphabetic character representing an allowable kind. Only spec codes may appear in the string and none more than once. The string defines all allowable kinds for this parameter. The spec codes for each kind of actual parameter are:

switch	W	sWitch
string	G	strinG
label	L	Label
no type procedure	P	Procedure
typed procedure	F	Function
array	A	Array
constant	C	Constant
expression	X	eXpression
simple variable	V	Variable
subscripted variable	S	Subscripted

If the actual parameter is not of the allowable kind, the diagnostic PARAMETER KIND ERROR is produced.

examples:

```
SPEC    FORMAT, GA      . format items
SPEC    X3, FCXVS       . right hand side
```

15.4 FORMAL HANDLING MACROS

Any kind of actual parameter may be passed to the COMPASS subprogram, and subsequent handling of them as formals is the responsibility of the subprogram. None of the macros in this section is applicable to all kinds of parameters. The most efficient way to ensure correct execution is by use of the macros, KIND and SPEC. An alternative method is provided by optional checking within the macro itself to ensure that the parameter kind is consistent with the specified operation. This automatic checking is less efficient since it must be processed in line to maintain flexibility, and a single check may be repeated several times on the same parameter. All macros assume that the contents of register B4 are unchanged after entry to the subprogram.

15.4.1 VALUE

This macro is applicable to the following parameters only: typed procedure, constant, expression, simple variable, or subscripted variable. The parameter may be checked for kind, and the macro will cause the current value of the parameter to be computed.

```
VALUE    param, xreg, kind, check
```

param: {absolute address expression, or X register designator} indicates ordinal of required actual parameter.

Xreg: {X register designator} the designated X register will contain the normalized floating point value of the indicated actual.

kind: (optional, absolute address expression) numerical kind code of the actual, if known; this parameter may be omitted. It will be ignored if its value cannot be ascertained at assembly time. The presence of this parameter will produce a more efficient expansion.

check: (optional) if this parameter is present (non-blank) the actual will be tested for kind. Abnormal termination with a PARAMETER KIND ERROR diagnostic may occur at execution time.

Examples:

```
VALUE    3, X1,, CHECK evaluate third parameter
VALUE    X1, X2, 6  a constant
VALUE    B, X6
```

15.4.2 ASSIGN

This macro is applicable only to a parameter which is either a simple variable or a subscripted variable, and it will optionally check for these kinds. It assigns the normalized contents of an X register to a specified parameter.

```
ASSIGN    param, xreg, kind, check
```

param: (absolute address expression, or X register designator) see 15.4.1

Xreg: (X-register designator) X register contains the normalized floating point value to be assigned to specified actual.

kind: (optional, absolute address expression) see 15.4.1

check: (optional) see 15.4.1

Examples:

```
ASSIGN    ANSWER, X1, 8
ASSIGN    3, X7,, CHECK
ASSIGN    A, X0
```

15.4.3 ADDRESS

This macro is applicable to the following parameters only: a string, an array, a constant, a simple variable, or a subscripted variable. It will optionally check for these kinds. It calculates the address, or first word address, of the specified parameter.

```
ADDRESS    param, xreg, kind, check
```

param: (absolute address expression, or X register designator) see 15.4.1

X-reg: (X-register designator) X register will receive the address of a constant or variable, or the first word address of an array or string.

kind: (optional, absolute address expression) see 15.4.1

check: (optional) see 15.4.1

Examples:

```
ADDRESS   3, X1, 1          string
ADDRESS   A, X5,, CHECK
ADDRESS   X1, X2, 9         subscripted variable
```

15.4.4 LENGTH

This macro is applicable only to an array or a string parameter. Optionally it will check kind. It will calculate the number of elements in the array or the number of characters in a string.

```
LENGTH    param, xreg, kind, check
```

param: (absolute address expression, or X-register designator) see 15.4.1

Xreg: (X register designator) X register will receive the appropriate length of the actual parameter calculated in fixed point binary

kind: (optional, absolute address expression) see 15.4.1

check: (optional) see 15.4.1

A string is stored in the coefficient part of a word with zero exponent, eight characters per word, and the count will include at least six characters for the initial and terminal string quotes.

Examples:

```
LENGTH X, X2, 1        string
LENGTH X3, X4, 5       array
```

15.4.5 FWA

This macro provides the address of a parameter; it applies only to an array or a string, giving the first word address of the elements thereof. The expansion is more efficient than that of the ADDRESS macro.

```
FWA    param, xreg, check
```

param: (absolute address expression, or X-register designator) see 15.4.1

Xreg: (X register designator) X register will receive the first word address of the array or string

check: (optional) see 15.4.1.

Example:

```
FWA 4, X3, CHECK
FWA X1, X2
```

15.4.6 ORDER

This macro operates only an array parameters. It returns the order, or dimensionality of the array (the number of subscripts required to address the array in its original declaration).

ORDER param, xreg, check

param: {absolute address expression, or X register designator} see 15.4.1.

xreg: {X register designator} the X register will receive, in fixed point binary, the order of the specified actual array parameter

check: {optional} see 15.4.1.

Example:

```
ORDER   A, X2, CHECK
ORDER   X3, X7
```

15.4.7 GOTO

This macro applies only to a switch or a label parameter (or a designational expression), and will optionally check kind. It causes a transfer to a label outside the COMPASS subprogram in accordance with the rules applying to the ALGOL go to statement, and it can be used on the occurrence of abnormal conditions within the COMPASS procedure. If the actual parameter is a switch (not a switch element), the value of the index for the required switch element must be supplied in the macro call.

GOTO param, index, check

param: (absolute address expression, or X-register designator) see 15.4.1.

index: (optional, absolute address expression or X-register designator) if omitted, actual is assumed to be a label; and control will be transferred to that point. If present, the actual is assumed to be a switch; and the value or contents of index must be the subscript of the required switch element. It will be evaluated and transferred to accordingly.

check: (optional) the specified actual will be checked to ascertain whether it is a label or a switch depending on the absence or presence of index. In addition, if index is present a check will be performed to ascertain that its value does not produce an undefined switch designator. If check is omitted no checking code will be assembled.

15.4.8 DOPE

This macro operates only on array parameters. It translates the ALGOL array descriptor and information into a form which permits array access by positive integer subscripts. A new dope vector is produced in the form of a table giving the order of the array and its absolute dimensions. This information may be used to pass the array as a parameter to a FORTRAN coded subroutine.

p_i: (optional, relocatable address expression, or special) up to 6 parameters of this form are permitted. If the first character of p_i is an asterisk (e.g., *3) p_i is interpreted as a special formal. Otherwise p_i must be a relocatable address expression. It is assumed to be the address of a variable local to the COMPASS subprogram which provides an input value to the formal procedure or expects to receive a result from the procedure. This variable is passed with specification, real, as the ith parameter to the formal procedure. If p_i is a special formal, the leading asterisk is discarded and the remainder must constitute an absolute address expression, or X-register designator, the value of which is interpreted as a parameter ordinal indexing the original parameters of the COMPASS subprogram (exactly as param). The resulting actual parameter is transmitted directly to the formal procedure as the ith parameter thereof.

Example: XEQ PROC, 2, PARAM, *1

The PROC parameter of the COMPASS subprogram must be a 2-parameter procedure. It is invoked with the variable stored at location PARAM as its first, real, parameter; its second parameter is the first parameter (*1) of the COMPASS subprogram.

STANDARD CHARACTER SETS AND ALGOL CHARACTER SETS

Shading denotes additional characters referred to in Appendix C.

The character set selected when the system is installed should be compatible with the printers.

With an installation parameter, the installation keypunch format standard can be selected as 026 or 029; the installation parameter can also allow a user to override the standard: a user may select a keypunch mode for his input deck by punching 26 or 29 in columns 79 and 80 of his JOB card or any 7/8/9 end-of-record card. The mode remains set for the remainder of the job or until it is reset by a different mode selection on another 7/8/9 card.

The sample program in appendix B may be punched in the ALGOL Extended 62-character set (appendix C) as follows:

```
'BEGIN' 'INTEGER' I;
    I:=10;
L:  I:=I+1;
    OUTPUT(61,'('/,3D')',I);
    'BEGIN' 'ARRAY' A[-3*I:-I, I:2*I]; 'INTEGER'P,Q;
    'FOR' P:=-3*I 'STEP' 1 'UNTIL' -I 'DO'
    'FOR' Q:=I 'STEP' 1 'UNTIL' 2*I 'DO'
        A[P,Q]:=-P+100*Q;
A[-2*I,I+2]:=A[-2*I,I+2] + 10000;
    'FOR' P:=-3*I 'STEP' 1 'UNTIL' -I 'DO'
    'FOR' Q:=I 'STEP' 1 'UNTIL' 2*I 'DO'
    'IF' A[P,Q]¬= 100*Q-P 'THEN'
    'BEGIN' OUTPUT (61,'('/,5D')',A[P,Q]) 'END';
    'GOTO' L
    'END'
    'END'.,
        'EOP'
```

NOTE: wherever ' is used, the character intended is:

CDC 63 or 64 character set	≠	(8-4 punch)
CDC ASCII subset	"	(8-4 punch)

The character ' is not in the ALGOL character set.

ALGOL 48-CHARACTER SET†

Character	Card Punch	Character	Card Punch
A	12-1	Y	0-8
B	12-2	Z	0-9
C	12-3	0	0
D	12-4	1	1
E	12-5	2	2
F	12-6	3	3
G	12-7	4	4
H	12-8	5	5
I	12-9	6	6
J	11-1	7	7
K	11-2	8	8
L	11-3	9	9
M	11-4	+	12
N	11-5	-	11
O	11-6	*	11-8-4
P	11-7	/	0-1
Q	11-8	=	8-3
R	11-9	(	0-8-4
S	0-2	)	12-8-4
T	0-3	.	12-8-3
U	0-4	,	0-8-3
V	0-5	'	8-4
W	0-6	$	11-8-3
X	0-7	␣ (space)	␣††

†Standard character sets follow.

††Blank column

SCOPE 3.3

CDC 64-CHARACTER SET

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	:†	8-2	8-2	00*	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	-	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	%	8-6	0-8-4	16
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	∨	11-0 or 11-8-2	11-0 or 11-8-2	52
27	W	0-6	0-6	26					
30	X	0-7	0-7	27	67	∧	0-8-7	12	37
31	Y	0-8	0-8	30	70	↑	11-8-5	8-4	55
32	Z	0-9	0-9	31	71	↓	11-8-6	0-8-7	56
33	0	0	0	12	72	<	12-0 or 12-8-2	12-0 or 12-8-2	72
34	1	1	1	01					
35	2	2	2	02	73	>	11-8-7	0-8-6	57
36	3	3	3	03	74	≤	8-5	12-8-4	15
37	4	4	4	04	75	≥	12-8-5	0-8-2	75
					76	¬	12-8-6	11-8-7	76
					77	;(semicolon)	12-8-7	11-8-6	77

†This character is lost on even parity magnetic tape.

*Since 00 cannot be represented on magnetic tape, it is converted to BCD 12. On input, it will be translated to display code 33 (number zero).

SCOPE 3.3

ASCII 64-CHARACTER SUBSET*

Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code	Display Code	Character	Hollerith (026)	Hollerith (029)	ASCII Code
00	: †	8-2	8-2	072	40	5	5	5	065
01	A	12-1	12-1	101	41	6	6	6	066
02	B	12-2	12-2	102	42	7	7	7	067
03	C	12-3	12-3	103	43	8	8	8	070
04	D	12-4	12-4	104	44	9	9	9	071
05	E	12-5	12-5	105	45	+	12	12-8-6	053
06	F	12-6	12-6	106	46	-	11	11	055
07	G	12-7	12-7	107	47	*	11-8-4	11-8-4	052
10	H	12-8	12-8	110	50	/	0-1	0-1	057
11	I	12-9	12-9	111	51	(	0-8-4	12-8-5	050
12	J	11-1	11-1	112	52	)	12-8-4	11-8-5	051
13	K	11-2	11-2	113	53	$	11-8-3	11-8-3	044
14	L	11-3	11-3	114	54	=	8-3	8-6	075
15	M	11-4	11-4	115	55	blank	no punch	no punch	040
16	N	11-5	11-5	116	56	, (comma)	0-8-3	0-8-3	054
17	O	11-6	11-6	117	57	. (period)	12-8-3	12-8-3	056
20	P	11-7	11-7	120	60	#	0-8-6	8-3	043
21	Q	11-8	11-8	121	61	' (apostrophe)	8-7	8-5	047
22	R	11-9	11-9	122	62	!	0-8-2	12-8-7	041
23	S	0-2	0-2	123	63	%	8-6	0-8-4	045
24	T	0-3	0-3	124	64	" (quote)	8-4	8-7	042
25	U	0-4	0-4	125	65	_ (underline)	0-8-5	0-8-5	137
26	V	0-5	0-5	126	66	]	11-0 or 11-8-2	11-0 or 11-8-2	135
27	W	0-6	0-6	127					
30	X	0-7	0-7	130	67	&	0-8-7	12	046
31	Y	0-8	0-8	131	70	@	11-8-5	8-4	100
32	Z	0-9	0-9	132	71	?	11-8-6	0-8-7	077
33	0	0	0	060	72	[	12-0 or 12-8-2	12-0 or 12-8-2	133
34	1	1	1	061					
35	2	2	2	062	73	>	11-8-7	0-8-6	076
36	3	3	3	063	74	<	8-5	12-8-4	074
37	4	4	4	064	75	\	12-8-5	0-8-2	134
					76	^ (circumflex)	12-8-6	11-8-7	136
					77	; (semicolon)	12-8-7	11-8-6	073

†This character is lost on even parity magnetic tape.

*BCD representation is used when data is recorded on even parity magnetic tape. In this case, the octal BCD/display code correspondence is the same as for the CDC 64-character set.

SCOPE 3.3

CDC 63-CHARACTER SET

Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD	Display Code	Character	Hollerith (026)	Hollerith (029)	External BCD
00	(none)†			16	40	5	5	5	05
01	A	12-1	12-1	61	41	6	6	6	06
02	B	12-2	12-2	62	42	7	7	7	07
03	C	12-3	12-3	63	43	8	8	8	10
04	D	12-4	12-4	64	44	9	9	9	11
05	E	12-5	12-5	65	45	+	12	12-8-6	60
06	F	12-6	12-6	66	46	-	11	11	40
07	G	12-7	12-7	67	47	*	11-8-4	11-8-4	54
10	H	12-8	12-8	70	50	/	0-1	0-1	21
11	I	12-9	12-9	71	51	(	0-8-4	12-8-5	34
12	J	11-1	11-1	41	52	)	12-8-4	11-8-5	74
13	K	11-2	11-2	42	53	$	11-8-3	11-8-3	53
14	L	11-3	11-3	43	54	=	8-3	8-6	13
15	M	11-4	11-4	44	55	blank	no punch	no punch	20
16	N	11-5	11-5	45	56	, (comma)	0-8-3	0-8-3	33
17	O	11-6	11-6	46	57	. (period)	12-8-3	12-8-3	73
20	P	11-7	11-7	47	60	≡	0-8-6	8-3	36
21	Q	11-8	11-8	50	61	[	8-7	8-5	17
22	R	11-9	11-9	51	62	]	0-8-2	12-8-7	32
23	S	0-2	0-2	22	63	:(colon)†	8-2	8-2	00*
24	T	0-3	0-3	23	64	≠	8-4	8-7	14
25	U	0-4	0-4	24	65	→	0-8-5	0-8-5	35
26	V	0-5	0-5	25	66	∨	11-0 or 11-8-2	11-0 or 11-8-2	52
27	W	0-6	0-6	26					
30	X	0-7	0-7	27	67	∧	0-8-7	12	37
31	Y	0-8	0-8	30	70	↑	11-8-5	8-4	55
32	Z	0-9	0-9	31	71	↓	11-8-6	0-8-7	56
33	0	0	0	12	72	<	12-0 or 12-8-2	12-0 or 12-8-2	72
34	1	1	1	01					
35	2	2	2	02	73	>	11-8-7	0-8-6	57
36	3	3	3	03	74	≤	8-5	12-8-4	15
37	4	4	4	04	75	≥	12-8-5	0-8-2	75
					76	¬	12-8-6	11-8-7	76
					77	;(semicolon)	12-8-7	11-8-6	77

†When the 63-Character Set is used, the punch code 8-2 is associated with display code 63, the colon. Display code 00_8 is not included in the 63-Character Set and is not associated with any card punch. The 8-6 card punch (026 keypunch) and the 0-8-4 card punch (029 keypunch) in the 63-Character Set are treated as blank on input.

*Since 00 cannot be represented on magnetic tape, it is converted to BCD 12. On input, it will be translated to display code 33 (number zero).

SCOPE 3.4/KRONOS 2.1§
STANDARD CHARACTER SETS

CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code	CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code
:†	:	00†	8-2	00	8-2	3A	6	6	41	6	06	6	36
A	A	01	12-1	61	12-1	41	7	7	42	7	07	7	37
B	B	02	12-2	62	12-2	42	8	8	43	8	10	8	38
C	C	03	12-3	63	12-3	43	9	9	44	9	11	9	39
D	D	04	12-4	64	12-4	44	+	+	45	12	60	12-8-6	2B
E	E	05	12-5	65	12-5	45	-	-	46	11	40	11	2D
F	F	06	12-6	66	12-6	46	*	*	47	11-8-4	54	11-8-4	2A
G	G	07	12-7	67	12-7	47	/	/	50	0-1	21	0-1	2F
H	H	10	12-8	70	12-8	48	(	(	51	0-8-4	34	12-8-5	28
I	I	11	12-9	71	12-9	49	)	)	52	12-8-4	74	11-8-5	29
J	J	12	11-1	41	11-1	4A	$	$	53	11-8-3	53	11-8-3	24
K	K	13	11-2	42	11-2	4B	=	=	54	8-3	13	8-6	3D
L	L	14	11-3	43	11-3	4C	blank	blank	55	no punch	20	no punch	20
M	M	15	11-4	44	11-4	4D	, (comma)	, (comma)	56	0-8-3	33	0-8-3	2C
N	N	16	11-5	45	11-5	4E	. (period)	. (period)	57	12-8-3	73	12-8-3	2E
O	O	17	11-6	46	11-6	4F	≡	#	60	0-8-6	36	8-3	23
P	P	20	11-7	47	11-7	50	[	[	61	8-7	17	12-8-2	5B
Q	Q	21	11-8	50	11-8	51	]	]	62	0-8-2	32	11-8-2	5D
R	R	22	11-9	51	11-9	52	%††	%	63	8-6	16	0-8-4	25
S	S	23	0-2	22	0-2	53	≠	" (quote)	64	8-4	14	8-7	22
T	T	24	0-3	23	0-3	54	→	_ (underline)	65	0-8-5	35	0-8-5	5F
U	U	25	0-4	24	0-4	55	∨	!	66	11-0 or 11-8-2†††	52	12-8-7 or 11-0†††	21
V	V	26	0-5	25	0-5	56							
W	W	27	0-6	26	0-6	57	∧	&	67	0-8-7	37	12	26
X	X	30	0-7	27	0-7	58	↑	' (apostrophe)	70	11-8-5	55	8-5	27
Y	Y	31	0-8	30	0-8	59	↓	?	71	11-8-6	56	0-8-7	3F
Z	Z	32	0-9	31	0-9	5A	<	<	72	12-0 or 12-8-2†††	72	12-8-4 or 12-0†††	3C
0	0	33	0	12	0	30							
1	1	34	1	01	1	31	>	>	73	11-8-7	57	0-8-6	3E
2	2	35	2	02	2	32	≤	@	74	8-5	15	8-4	40
3	3	36	3	03	3	33	≥	\	75	12-8-5	75	0-8-2	5C
4	4	37	4	04	4	34	≠	^ (circumflex)	76	12-8-6	76	11-8-7	5E
5	5	40	5	05	5	35	; (semicolon)	; (semicolon)	77	12-8-7	77	11-8-6	3B

†Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

††In installations using the CDC 63-graphic set, display code 00 has no associated graphic or Hollerith code; display code 63 is the colon (8-2 punch).

†††The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

§CDC 63-graphic set is standard for KRONOS.

SAMPLE PROGRAM B

The following program is in the exact form that is punched into the cards that comprise the source deck (the hardware language).

```
2-DIMENSIONAL ARRAY.
THIS PROGRAM DECLARES A SERIES OF ARRAYS OF EVER-INCREASING
DIMENSION.  THE ARRAY IS THEN FILLED WITH COMPUTED VALUES, ONE
OF WHICH IS ALTERED.  THE ALTERED VALUE IS THEN SEARCHED FOR
AND PRINTED.
THE PROGRAM HALTS WHEN THE DECLARED ARRAY SIZE EXCEEDS THE
AVAILABLE MEMORY.  WHEN THIS OCCURS, THE PROGRAM EXITS WITH
THE MESSAGE          STACK OVERFLOW              ON THE STANDARD
OUTPUT UNIT.
'BEGIN' 'INTEGER' I.,
  I..=10.,
L..I..=I+1.,
  OUTPUT(61, '('/,3D')',I).,
  'BEGIN' 'ARRAY' A(/-3*I..-I,I..2*I/)., 'INTEGER' P,Q.,
    'FOR' P..=-3*I 'STEP' 1 'UNTIL' -I 'DO'
      'FOR' Q..=I 'STEP' 1 'UNTIL' 2*I 'DO'
        A(/P,Q/)..=-P+100*Q.,
   A(/-2*I,I+2/)..=A(/-2*I,I+2/)+10000.,
   'FOR' P..=-3*I 'STEP' 1 'UNTIL' -I 'DO'
     'FOR' Q..=I 'STEP' 1 'UNTIL' 2*I 'DO'
       'IF' A(/P,Q/) 'NOT EQUAL' 100*Q-P 'THEN'
       'BEGIN' OUTPUT(61,'('/,5D')',A(/P,Q/)) 'END'.,
   'GOTO' L
  'END'
'END'.,
          'EOP'
```

CHARACTER REPRESENTATION OF ALGOL SYMBOLS C

Table 1. Character Representation of ALGOL Symbols

(The additional representations shown are all available with the CONTROL DATA character set, however some of the corresponding graphics do not exist in the ASCII character set. See Appendix A for differences and for the corresponding punch card codes.

ALGOL Symbol	48-Character Representation	Additional Representation	ALGOL Symbol	48-Character Representation
A-Z	A-Z		true	'TRUE'
a-z	(none)		false	'FALSE'
0-9	0-9		go to	'GO TO'
+	+		if	'IF'
-	-		then	'THEN'
×	*		else	'ELSE'
/	/		for	'FOR'
↑†	'POWER'	** or ↑	do	'DO'
÷	'/' or 'DIV'	//	step	'STEP'
>	'GREATER'	>	until	'UNTIL'
≥	'NOT LESS'	≥	while	'WHILE'
=	= or 'EQUAL'		comment	'COMMENT'
≠	'NOT EQUAL'	¬=	begin	'BEGIN'
≤	'NOT GREATER'	≤	end	'END'
<	'LESS'	<	own	'OWN'
∧	'AND'	∧	Boolean	'BOOLEAN'
∨	'OR'	∨	integer	'INTEGER'
≡	'EQUIV'	≡	real	'REAL'
¬	'NOT'	¬	array	'ARRAY'
⊃	'IMPL'		switch	'SWITCH'
.	.		procedure	'PROCEDURE'
,	,		string	'STRING'
:	..	:	label	'LABEL'
;	.,	;	value	'VALUE'
10	'	→	code ††	'CODE'
␣	␣ (space)		eop ††	'EOP'
(	(			
:=	.= or ..=	:=		
)	)			
[	(/	[		
]	/)	]		
‘	'('			
’	')'			

†In a format string, ↑ may be represented by an asterisk.

††Not defined in the ALGOL-60 Revised Report; code is defined in Section 5.4.1, Chapter 2; eop in Chapter 4.

INTERACTIVE ALGOL UNDER INTERCOM† SCOPE OPERATING SYSTEM

D

With terminal access provided by INTERCOM, the ALGOL user has a debugging facility available in two distinct parts: a syntax analyzer, which is used before the program is compiled, and a breakpoint facility, which is used after compilation while the program is being executed.

While the source program is being typed in or modified, the syntax analyzer checks each line and responds with a diagnostic message when an error is detected. Syntax checking can be turned off and on dynamically, so that several statements may be diagnosed at one time.

To select the breakpoint facility, the user compiles with the H option and then executes his program interactively. Execution can be suspended and control given to the user so that he can check intermediate values, make decisions, and then resume execution until the next breakpoint is reached.

The user controls where execution is suspended by placing BRKPTI statements in his source program and specifying, at execution time, the class of breakpoints he wants the system to honor. After the program is debugged, the user can execute the program, having the breakpoints ignored; or he can recompile, having the compiler ignore the BRKPTI statements. In addition to breakpoints explictly specified in the source program, breakpoints can be activated at procedure calls, and at the beginning of each block.

ENTERING AND MODIFYING FILE: ALGEDIT

The command ALGEDIT is used to get line-by-line syntax checking when a program is being entered or modified.

Execept for the addition of three commands, ALGEDIT has all the features of EDITOR, as described in the INTERCOM Manual. Underlined characters indicate acceptable abreviations of these ALGEDIT commands:

ANALYZE

CHECK

NOCHECK

The default FORMAT parameters for ALGEDIT are:

Character count = 72

Tab character = $

Tab columns = 7 10 13 16 19

†The interactive features described in this appendix are not available under KRONOS.

If the character count is set to a value greater than 72, or a line of greater than 72 characters is encountered by ALGEDIT, the following message will be printed and any excess characters will be ignored:

```
LINE LENGTH RESET TO 72
```

The ALGOL diagnostics given by ALGEDIT are the same as those provided by the ALGOL compiler when the Z option is specified. A list of the possible diagnostics appears at the end of this appendix.

ANALYZE

This command turns off the CHECK mode and causes an immediate syntax check beginning at the first statement and proceeding through the last. The list of any diagnostic messages is followed by the standard EDITOR pair of dots.

If the Edit file is empty, this message is printed:

```
ERR- NO INFORMATION IN EDIT FILE
```

CHECK

This command activates line-by-line syntax checking which remains effective until an ANALYZE or NOCHECK command is received. Checking resumes from where it last was performed. As a line is sent from the terminal, the text is analyzed and checked for consistency with the lines previously entered. If the line number is greater than or equal to the previous one, analysis is performed on that line. If the line number is smaller than the previous one, analysis will be reinitialized from the first entered statement and continue through the last new line.

NOCHECK

This command stops the line-by-line syntax checking activated by the previous CHECK command.

The CHECK or NOCHECK status is not altered by the use of LIST, SAVE, RESEQ, RUN, or EDIT, commands.

COMPILATION AND EXECUTION IN INTERACTIVE MODE

Compile Time Option

On the ALGOL control card the H option and the M option are needed when execution is to be interactive. The H option is not compatible with the S and R options, and it cannot be used in segmented mode. The following message will appear if R and S options are used with H.

```
ALGOL CON-CARD ERROR
    H DELETED
```

Execution Time Option

To execute with interactive debugging, option H is required on one of the following SCOPE control cards; LGO, EXECUTE, or program name. This option is switched off when the program is run in batch mode or when the program was compiled without the H option.

Also, this option can be specified along with any CHANNEL cards as the first data on the standard input device. The format is:

```
OPTIONS,H
```

If a request is made to run a program with the H option when this option has not been given at compile time, the program will be terminated and the following message will be displayed:

```
RECOMPILE PROGRAM WITH H OPTION.
```

The H option is switch off when the program execution is in batch mode or when the user explicitly turns it off when he gets control at a breakpoint. In either case, it is impossible to return to the interactive debugging mode of operation during the current execution.

When the user's program starts executing, the following messages will be displayed:

```
YOU ARE IN INTERACTIVE DEBUGGING MODE FOR EXECUTION OF YOUR PROGRAM
  INDICATE WHERE YOU WANT CONTROL
```

The user's response should indicate the breakpoints at which he chooses to stop execution.

The user can select one or more of the options listed below. When several active breakpoints are selected, the options are separated by commas.

ALL	All breakpoints both implicit and explicit
AB	Abort
BB	Block by block
EE	From end block to end block
II	From BRKPTI to BRKPTI
PP	Procedure by procedure
NB	Next block
NE	Next end block
NI	Next BRKPTI
NP	Next procedure
PP	From call procedure to call procedure
B=xx	xx is the numberic level of the block selected.
P=xx	xx is the mnemonic name of the procedure selected.
X=yy	yy is the integer control value.
OFF	Checking off (ignore all breakpoints)

To begin or resume program execution, the user must send the 'GO' command. Initially the default values are ALL and X=0.

User Actions at Interrupt

During execution, one of the following message will inform the user that a breakpoint occurred (xx is the static blcok level number rather than the dynamic level number):

YOU NOW HAVE CONTROL { AT START OF BLOCK / AT END OF BLOCK / AT END OF PROCEDURE / AT CALL OF PROCEDURE } xx LINE yy

Requested Data

The user can send the following commands:

'ALL'

or name of a specific variable

Values are requested by specifying variable names either on one line separated by commas or on separate lines. To reference a specific element of the array, the user can enter the array name with subscript values as integer constants within parentheses. However, all elements of a row, column, plane ... of an array can be requested by using an asterisk(*) for the corresponding subscript. The upper and lower bounds are obtained by sending only the array name. The response will be the variable name with its value. However, if the number of variables should exceed the capacity of a CRT screen, the user must request the screen to be reloaded by typing the send key.

The user can terminate dumping of values by sending 'END'. This command returns the usual breakpoint message to the user. For a program check, the user will not get an automatic dump; but he will get control at the next requested breakpoint. Then, the normal procedure will apply, except the program will be terminated after the 'GO' request has been sent.

The following restrictions apply:

No data can be requested by name

No function procedures

No switches

No labels

No strings

The user can modify the value of any variable by typing the name of the variable followed by the assign (: =) and the value to be assigned. The value must be a legal ALGOL constant for the variable type.

Example:

```
A       : =  3.4

I       : =  1

B       : =  'TRUE'

AR [2,3]: =     - 4.5
```

'SET'

This command allows the user to modify the breakpoint status or the control value by bringing back the display:

```
INDICATE WHERE YOU WANT CONTROL
```

'LIST'

This command produces a list of the breakpoint options.

'TIME'

This command provides the current time and date in the following format:

```
TIME = xx:xx:xx                    DATE = xx:xx:xx
```

'FL'

With this command, the user obtains the field length at the time the command is used. The format is:

```
FL USED = value in octal           UNUSED = value in octal
```

'CLOCK'

With this command, the user obtains the CPU time he has used for the currently executing ALGOL programs. The format is:

CPU = decimal value SECONDS

'GO'

This command restarts program execution.

'END'

This command stops the display of data when several screens or lines are involved. If the user sends a message that is not correct, the following messages will appear:

I DON'T UNDERSTAND. TRY AGAIN.

If the user requests data that is either unknown or outside the block range of his ALGOL program:

NAME UNKNOWN. TRY AGAIN.

When several commands appear on the same line, they must be separated with commas; and the following restrictions must be observed:

'GO' must not be followed by another command.

'LIST' must not be followed by another command.

If more than 15 data values are requested, the remaining commands will be ignored. After the 15 data values are printed, the user must enter either 'END' or a carriage return.

'ST'

This command provides a list of currently active breakpoints.

SETTING BREAKPOINTS IN SOURCE PROGRAM

A breakpoint can be created in the source text by inserting the BRKPTI standard procedure. In batch mode or without the H execution option, this procedure is not effective; when active, it returns control to the user at the breakpoints. BRKPTI is made active if the user has selected at least one of the following breakpoint control parameters:

ALL I I N I

PP NP

The BRKPTI procedure is written into the source program as:

; BRKPTI (n) ;

n is an integer parameter whose value is to be displayed at the remote terminal when the breakpoint occurs.

The n parameter of the BRKPTI procedure is compared to a control value X. If X $\leq$ n the breakpoint is active; if X > n the breakpoint is inactive. At the beginning of run time, the control value X is set to zero; but the user can change this value when he sets the breakpoint status.

Sample ALGOL compilation and execution at a TTY terminal:

It illustrates typical use of the interactive features of ALGOL 3.1. The descriptions with the example assume the knowledge of ALGOL 60 Version 3. The INTERCOM 3 Reference Manual, Publication No. 60258000, should be consulted for more information regarding the use of INTERCOM. This example is only one of many ways the user may choose to use the interactive facility provided through INTERCOM and ALGOL. Under SCOPE 3.4, the " must be used in place of the '.

```
CONTROL DATA INTERCOM 3.0
DATE  05/18/72
TIME  08.12.22.
LOGIN.
ENTER USER NAME-SVLDOC
[redacted] ENTER PASSWORD-

 05/18/72    LOGGED IN AT  08.13.10.
             WITH USER-ID AJ
             EQUIP/PORT 51/17
COMMAND- ASSETS
ASSETS OF AJ AT  08.13.30.
EQUIP/PORT    51/17
FILE QUOTA       20
FILES IN USE      1
MAX FL         77700
TIME LIMIT      0500
CP TIME          .044
PP TIME         2.939
COMMAND- FILES
--PRIVATE FILES--
  $OUTPUT

COMMAND- ALGEDIT.

..CHECK
..CREATE

   100:$'BLGIN' 'INTEGER' IJ., 'REAL' X,Y., 'BOOLEAN' B.,
   110:$'PROCEDURE' P(P1,P2)., 'VALUE' P2., 'INTEGER' P1,P2..
   120:$$'BEGIN'
   130:$$$'ARRAY'A)/1..P1,1..P2/)..,
   LINE     130     ) ILLEGAL IN ARRAY DECLARATION

   140=:130:$$$'ARRAY'A(/1..P1,1..P2/)..,
   140:$$$'INTEGER' X,Y.,
   150:$$$P1..:P1+1.,
   160:$$$'FOR' X..:1 'STEP' 1 'UNTIL' P2 ' 'DO'
   LINE     160     UNRECOGNIZABLE COMPOUND DELIMETER ''

   170=:160:$$$'FOR' X..:1 'STEP' 1 'UNTIL' P2  'DO'

   170:$$$'FOR' Y ..: 1 'STEP' 1 'UNTIL' P2 'DO' A(/X,Y/) ..: X+Y/10.,
   180:$$$BRKPTI(1).,


   190:$$'END' P.,
   200:$1..:0., J..:1., X..:2., Y..:3., B..: 'TRUE' .,
   210:$$$L.. BRKPTI(0).,
   220:$$$$P(J,5).,
   230:$$$$'GOTO' L.,
   240:$'END'
   250:$$$$'EOP'
   260:E

..SAVE TEST

..ANALYZE
```

Log in

Verify that available field length is approximately 70000 octal words.

Verify no existing files that might cause problems because of duplicate file names

Call ALGEDIT

Interactively CHECK each line

ALGEDIT diagnostic

Reenter line 130 correctly

Reenter line 160 correctly

Enter ALGOL program using $ to set tabs at locations 7, 10, 13, and 16

Exit CREATE mode

Save program as private file with name of TEST

Check syntax for entrie program

```
   LINE     250    FINIS GEN. BY EOR-CARD
   LINE     100    PROGRAM BEGINS        (MESSAGE)
   LINE     240    PROGRAM ENDS          (MESSAGE)
   LINE     250    SOURCE DECK ENDS      (MESSAGE)
```

These lines are informative only and do not indicate errors

```
..BYE,BYE
```

Exit CREATE mode

```
COMMAND- REDUCE OFF
```

REDUCE,OFF is necessary since ALGOL requires certain locations for execution of object programs.

```
COMMAND- CONNECT INPUT,OUTPUT.
```

Input and output of compiler is directed to terminal

```
COMMAND- ALGEDIT.
```

Reentering to COMPILE from ALGEDIT

```
..EDIT TEST
```

Define file TEST to be edited

```
..RUN ALGOL
```

Try compiling to test a complete compilation

```
  JOB COMPILING
        0       186

CPU ABORT
```

Compilation was not successful

```
..ERRORS ALGOL
```

Use ERRORS to get diagnostics printout. Listing of ALGOL diagnostics and related source line. ALGEDIT line number is with the source line; and the LINE is the compiler generated line number

```
 ALGOL 60     PROGRAM   XXALGOL

000100          'BEGIN' 'INTEGER' IJ., 'REAL' X,Y., 'BOOLEAN' B.,
        LINE    0           PROGRAM BEGINS        (MESSAGE)
     1
000240          'END'
        LINE    14          PROGRAM ENDS          (MESSAGE)
     1
000250          'EOP'
        LINE    15          SOURCE DECK ENDS      (MESSAGE)
     1
                'EOP'
        LINE    8           DECLARATION MISSING FOR.. I
     6
000200          I..=0., J..=1., X..=2., Y..=3., B..= 'TRUE' ..
        LINE    10          DECLARATION MISSING FOR.. J
```

Comma missing between I and J in first line of program caused error. Other diagnostics would result from same error

```
MA
USER ABORT
```

On detection of error, user aborts ERRORS program

```
..100=$'BEGIN' 'INTEGER' I,J., 'REAL' X,Y., 'BOOLEAN' B.,
```

Enter correct line.

```
XYZ
```

Operator was confused and typed a couple carriage-returns then XYZ

```
 XYZ NOT IN PP LIB
..CHECK
..LIST ALL
```

List new version. Note that $'s have been replaced by correct indentation

```
   100=      'BEGIN' 'INTEGER' I,J., 'REAL' X,Y., 'BOOLEAN' B.,
   110=      'PROCEDURE' P(P1,P2)., 'VALUE' P2., 'INTEGER' P1,P2.,
   120=         'BEGIN'
   130=            'ARRAY'A(/1..P1,1..P2/).,
   140=            'INTEGER' X,Y.,
   150=            P1..=P1+1.,
   160=            'FOR' X..=1 'STEP' 1 'UNTIL' P2  'DO'
   170=            'FOR' Y ..= 1 'STEP' 1 'UNTIL' P2 'DO' A(/X,Y/) ..= X
+Y/10.,
   180=            BRKPTI(I).,
   190=         'END' P.,
   200=      I..=0., J..=1., X..=2., Y..=3., B..= 'TRUE' .,
   210=            L.. BRKPTI(0).,
   220=               P(J,5).,
   230=               'GOTO' L.,
   240=      'END'
   250=               'EOP'

..SAVE TEST OVERWRITE

..BYE,BYE

COMMAND- CONNECT INPUT,OUTPUT.
```

Reconnect INPUT and OUTPUT because ERRORS ALGOL was aborted. This routine normally disconnects OUTPUT on entry and then resets it on exit

```
COMMAND- REWIND,LGO.
```

Ensure a fresh start

```
COMMAND- ALGOL,S=0,R=0,L,H,M,X,I=TEST.
```

Compile the file TEST, with H option. This example explicitly sets parameters to produce complete listings. (parameters available are discussed in section 6)

```
ALGOL-60      (3.1)                                                    05/
18/72   08.29 HRS          PAGE   1

   00**        'BEGIN' 'INTEGER' I,J., 'REAL' X,Y., 'BOOLEAN' B.,
        000100
              'PROCEDURE' P(P1,P2)., 'VALUE' P2., 'INTEGER' P1,P2.,
        000110
                 'BEGIN'
        000120
                    'ARRAY'A(/1..P1,1..P2/).,
        000130
                    'INTEGER' X,Y.,
        000140
                    P1..=P1+1.,
        000150
                    'FOR' X..=1 'STEP' 1 'UNTIL' P2  'DO'
        000160
                    'FOR' Y ..= 1 'STEP' 1 'UNTIL' P2 'DO' A(/X,Y/) ..=
 X+Y/10.,000170
                    BRKPTI(I).,
        000180
                 'END' P.,
        000190
   10**      I..=0., J..=1., X..=2., Y..=3., B..= 'TRUE' .,
        000200
                    L.. BRKPTI(0).,
        000210
                       P(J,5).,
        000220
                       'GOTO' L.,
        000230
              'END'
        000240
                       'EOP'
        000250
                 FINIS  GEN.BY EOR-CARD
```

Compiler generated listing of program. Because TTY has only 72 print positions in a line, longer lines from ALGOL compiler are folded onto a second line

```
ALGOL-60      (3.1)                    XXALGOL                          05/
18/72   08.29 HRS          PAGE   1
                                                IDENTIFIER  TABLE

                  LEVEL LINE          NAME        TYPE  DM/P LINE  REF
     NAME        TYPE  DM/P LINE  REF

  BEGIN  BLOCK          0    0
                                        B           B              0    6
     I           I              0   12
                                        J           I              0   11
     L             L           11    4
                                        P             P      2     1    5
     X           R              0   10
                                        Y           R              0    7
  END OF BLOCK          0   14
  PROC.  P              1    1
                                        P1          I   S             - 4
     P2          I   V             - 5
                                        A           R A      2     3   14
     P1          I   S             - 4
                                        P2          I   V             - 5
     X           I              4    7
                                        Y           I              4    6
     *FOR LABEL*                6    5
                                        *FOR LABEL*                7    4
  END OF P              1    9
 NORMAL END OF MAP
```

Compiler generated MAP on a 72-character line

```
ALGOL-60      (3.1)                    XXALGOL                          05/
18/72   08.29 HRS            PAGE    1

        LINE    0          PROGRAM BEGINS       (MESSAGE)
        LINE   14          PROGRAM ENDS         (MESSAGE)
        LINE   15          SOURCE DECK ENDS     (MESSAGE)
```

Compiler summary.

```
                THE FOLLOWING CONTROL CARD OPTIONS ARE ACTIVE      F,
H,I,L,M,X
        0        543
 ILLEGAL I/O REQUEST
  FILE NAME  OUTPUT
  FET ADDRESS  002057
  CIO CODE NOT DEFINED ON DEVICE
COMMAND- REWIND,LGO.

COMMAND- FILES
```

List files generated by compiler

```
--PRIVATE FILES--
  SETFILE     LGO       $ZZZZ09     $OUTPUT
 $INPUT       TEST

COMMAND- LGO.
```

Execute program just compiled

Default channel number associations typed out by ALGOL

```
CHANNEL,60=INPUT,P80,R
CHANNEL,61=OUTPUT,P136,PP60,R
```

```
OPTIONS,H
```

User wants breakpoints honored

```
OPTIONS,H
CHANNEL,END
```

```
CHANNEL,END
```

Terminator for object time parameters to ALGOL system

```
YOU ARE IN INTERACTIVE DEBUGGING MODE FOR EXECUTION OF YOUR PROGRAM
INDICATE WHERE YOU WANT CONTROL 'LIST'

'LIST'
```

Use 'LIST' to obtain a list of the available commands

```
 ALL      ALL IMPLICIT AND EXPLICIT BREAKPOINTS
 AB       ABORT CURRENT EXECUTION
 BB       FROM START OF BLOCK TO START OF BLOCK
 EE       FOR BLOCK OR PROC,FROM END TO END
 II       FROM BRKPTI TO BRKPTI
 NB       NEXT START OF BLOCK
 NE       NEXT END OF BLOCK OR PROCEDURE
 NI       NEXT CALL OF BRKPTI
 NP       NEXT CALL OF PROCEDURE
 PP       FROM CALL PROCEDURE TO CALL PROCEDURE
 OFF      MAKES ALL BREAKPOINTS INACTIVE
 B= XX    BLOCK XX

TYPE 'END' OR BLANK + CR
```

User typed blank and then a carriage-return

```
 P= YYYY  PROCEDURE YYYY
 X= ZZ    RESET VALUE OF X TO ZZ
 'CLOCK'  PROVIDES CPU TIME USED
 'END'    STOPS THE CURRENT OUTPUT
 'FL'     PROVIDES USED/UNUSED FIELD LENGTH
 'GO'     RESUME PROGRAM EXECUTION
 'LIST'   PROVIDES THIS LIST
 'SET'    SELECTS CONTROL MODE
 'ST'     PROVIDES LIST OF SELECTED BREAKPOINTS
 'TIME'   PROVIDES TIME AND DATE
DEAD?
          DEAD? UNRECOGNIZABLE. ALL THAT FOLLOWS HAS BEEN IGNORED

TRY AGAIN
'STATUS'
         'STATUS' UNRECOGNIZABLE. ALL THAT FOLLOWS HAS BEEN IGNORED
```

```
'STATUS'
```

User attempts to obtain status by entering full word 'STATUS'

```
TRY AGAIN
'ST'
STATUS
  ALL,X=0
OFF,BB,'GO'
```

```
'ST'
```

Abbreviation 'ST' must be used to obtain status

```
OFF,BB,'GO'
```

User sets breakpoint parameters

```
        YOU NOW HAVE CONTROL AT START OF BLOCK     XXALGOL   LINE    0
REQUESTED DATA              I,J,X,Y,B
I,J,X,Y,B
                  I = -5.9604470688300'-008        DECLARED  LINE    0
                  J = +2.7957679633036'-257        DECLARED  LINE    0
                  X = +4.3555697992251'-279        DECLARED  LINE    0
                  Y = +6.8552147615436'-202        DECLARED  LINE    0
                  B = 'TRUE'                       DECLARED  LINE    0
REQUESTED DATA              'SET'
'SET'
INDICATE WHERE YOU WANT CONTROL                OFF,II
OFF,II                      'GO'
'GO'

       YOU NOW HAVE CONTROL AT CALL OF PROCEDURE BRKPTI    LINE   11


 X=0 N=0
REQUESTED DATA              'CLOCK'
'CLOCK'
        CPU TIME =          0.415
REQUESTED DATA              'TIME'
'TIME'

           TIME=  08.45.42.   DATE = 05/18/72
REQUESTED DATA              'FL'
'FL'

       FL USED =015636                UNUSED =022142
REQUESTED DATA              I,J,X,Y,B
I,J,X,Y,B
                  I =  0                           DECLARED  LINE    0
                  J =  1                           DECLARED  LINE    0
                  X = +2.0000000000000'+000        DECLARED  LINE    0
                  Y = +3.0000000000000'+000        DECLARED  LINE    0
                  B = 'TRUE'                       DECLARED  LINE    0
REQUESTED DATA              L,P
L,P
NO VALUE GIVEN FOR           L
NO VALUE GIVEN FOR           P
REQUESTED DATA              A
A
NO VALUE GIVEN FOR           A
REQUESTED DATA              TRICK
TRICK
       IDENTIFIER      TRICK NOT DECLARED OR INVALID IDENTIFIER STRING

REQUESTED DATA              'SET'
'SET'
INDICATE WHERE YOU WANT CONTROL                P:P
P:P                         'GO'
'GO'

       YOU NOW HAVE CONTROL AT CALL OF PROCEDURE P         LINE   12
REQUESTED DATA              A
A
NO VALUE GIVEN FOR           A
REQUESTED DATA              B,X
B,X
                  B = 'TRUE'                       DECLARED  LINE    0
                  X = -2.3282996362617'-010        DECLARED  LINE    4
REQUESTED DATA              'GO'
'GO'
         ARRAY BOUNDS ERROR

           THIS ERROR OCCURRED AFTER LINE    1  IN P

   THE GLOBAL VARIABLES ARE ..
UA,VALUE 015650       5725 015653 0000 015646
UV'      000000       0000 000000 0000 000000
LASTUSED 015657       1720 600000 0000 000000

            IN THE BLOCK ENTERED AT LINE    1  IN P


   THE FORMAL VARIABLES ARE ..
 000003  015626       1722 500000 0000 000000
 000002  015627       1722 500000 0000 000000
 000001  015630       2101 000000 0000 015624
 000000  015631       0002 776151 4004 015613

   THE LOCAL VARIABLES ARE ..
 000004  015640       1757 776250 4000 015634       1757 776263 4000 015
634       1720 400000 0000 000000       1721 400000 0000 000000
 000010  015644       1722 500000 0000 000000       .UNTIL.  000012  015
646       1722 600000 0000 000000       0000 000000 0000 000002
 000014  015650       5725 015653 0000 015646

          THIS BLOCK WAS CALLED FROM LINE   12  IN XXALGOL

            IN THE BLOCK ENTERED AT LINE    0  IN XXALGOL

   THE LOCAL VARIABLES ARE ..
 000004  015617       5757 776173 4000 015613       7747 776351 4000 015
613       4000 000000 0000 000000       1721 600000 0000 000000
 000010  015623       1721 400000 0000 000000       .UNTIL.  000012  015
625       0000 000000 0000 000000

          THIS BLOCK WAS CALLED FROM LINE    0  IN ALGORUN

CPU ABORT
COMMAND- LOGOUT.
CP TIME     3.171
PP TIME    61.218
CONNECT TIME    0 HRS. 39 MIN.
 05/18/72  LOGGED OUT AT 08.52.42.
```

Since values have not been given to these variables, they are of no value

Sample use of 'CLOCK' 'TIME' and 'FL'

Variables now have values

Object time error causes the following stack information to be printed (section 11)

M OPTION STORAGE ALLOCATION

E

Under the M option, the number of identifiers that can be handled successfully is relative to the field length selected. The normal field length provides adequate space for most programs to be compiled successfully under the M option. However, for programs containing an exceptionally large number of unique identifiers, the user should consult the following table of recommended field lengths.

Compilation field lengths (octal)	Max. No. of Identifiers (decimal)
30000	512
40000	1024
50000	1536
60000	2048
70000	2560
100000	3072
110000	3584

INDEX

Active
   'ST' Status of Active Breakpoints D-7
Aids
   Debugging Aids TRACE, SNAP, OFF 14-1
ALGEDIT
   Editing a File using ALGEDIT D-1
   ALGEDIT Tab Characters $ D-1
ALGOL
   ALGOL Character Representation C-1, 4-7
   ALGOL Control Cards and Options 6-1
   ALGOL 48 Character Set A-1
ALGOL-60
   ALGOL-60 Revised Report 2-2
Alpha
   String and Alpha Formats 3-9
Alphabetic
   Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49
ANALYZE
   ANALYZE Command for Syntax Checking D-2
Arithmetic
   Arithmetic, Boolean, and Designational Expressions 2-13
   Arithmetic Expression Syntax 2-17
Array
   Array Declarations and Dimensions 2-38
Arrays
   Variables, Arrays, and Subscripts 2-14
Assignment
   Assignment Statements 2-27

Basic
   Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8
Backus
   Backus Metalinguistic Notation for Syntax Description 2-7
Blank
   Comments and Blank Space 2-9
Blocks
   Compound Statements and Blocks 2-25
Boolean
   Arithmetic, Boolean, and Designational Expressions 2-13
   Boolean Expression Syntax 2-22
Bracket
   Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Breakpoint
   Breakpoint Selection D-4
   'SET' Breakpoints D-6
   'LIST' Breakpoint Options D-6
   'ST' Status of Active Breakpoints D-7
   Setting Breakpoints in Source Program BRKPTI(n) D-8

BRKPTI(n)
Setting Breakpoints in Source Program BRKPTI(n) D-8

Channel
Unit or Channel Numbers 3-15
Input-Output Channel Definition 7-1
Character
Character Strings 2-12
ALGOL Character Representation C-1, 4-7
ALGOL 48 Character Set A-1
Character Codes A-1
CHECK
CHECK Command for Interactive Syntax Checking D-2
Code
Non-ALGOL Code as Procedure Body 2-43
Coded
Linking Non-ALGOL Coded Subprograms or Procedures 15-1
Codes
Format Codes Summary 3-12
Character Codes A-1
Comments
Comments and Blank Space 2-9
COMPASS
COMPASS Coded Utility Macros 15-1
Compilation
Compilation and Source Program Definition 4-1
Interactive Compilation and Execution D-3
Compiler
Compiler and System Description 1-1
Compiler Output 5-1
Compiler Diagnostics and Error Messages 8-1
Compiler Description 9-1
Compound
Compound Statements and Blocks 2-25
Concepts
Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49
Conditional
Conditional or If Statements 2-29
Control
ALGOL Control Cards and Options 6-1
Conventions
Reference Manual Language Conventions and Notation 2-1
CPU
'CLOCK' CPU Time D-7

Date
'TIME' and Date D-6
Debugging
Debugging and Execution-Time Options 13-1
Debugging Aids TRACE, SNAP, OFF 14-1
Debugging Output 14-5
Interactive Debugging under INTERCOM D-1
Decimal
Numbers Integer, Decimal, and Exponent Parts 2-10
Deck
Source Deck Structures 4-4

Declarations
Type Declarations 2-36
Array Declarations and Dimensions 2-38
Switch Declarations 2-39
Procedure Declarations 2-40
Declarator
Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Definitions
Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49
Delimiter
Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8
Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Description
Compiler Description 9-1
Designational
Arithmetic, Boolean, and Designational Expressions 2-13
Designational Expression Syntax 2-24
Diagnostics
Compiler Diagnostics and Error Messages 8-1
Object-Time Diagnostics 8-9
Digit
Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8
Dimensions
Array Declarations and Dimensions 2-38
Display
'END' Interruption or Terminating Display of Data D-7
Dummy
Dummy Statements 2-29
Dump
Object-Time Dump 12-1

Editing
Editing a File using ALGEDIT D-1
End-of-file
Input-Output Errors, End-of-file End-of-Tape 3-44
End-of-Tape
Input-Output Errors, End-of-file End-of-Tape 3-44
ENTIER
Transfer Function ENTIER 2-17
Error
Compiler Diagnostics and Error Messages 8-1
Errors
Input-Output Errors, End-of-file End-of-Tape 3-44
Example
Sample Interactive Example under INTERCOM D-9
Execution
Interactive Compilation and Execution D-3
'GO' Resume Execution D-7
Debugging and Execution-Time Options 13-1
Exponent
Numbers Integer, Decimal, and Exponent Parts 2-10
Expressions
Arithmetic, Boolean, and Designational Expressions 2-13
Arithmetic Expression Syntax 2-17
Boolean Expression Syntax 2-22
Designational Expression Syntax 2-24
Extended
FORTRAN Extended and RUN Interface Macros 15-1

Field
'FL' Field Length D-6
File
Editing a File using ALGEDIT D-1
For
For Statement 2-31
Format
Format Strings 3-11
Format Codes Summary 3-12
Standard Format 3-12
Formats
Number Formats 3-2
String and Alpha Formats 3-9
FORTRAN
FORTRAN Extended and RUN Interface Macros 15-1
For-Loop
Implied For-Loop 3-29
Function
Function and Procedure Identifiers 2-15
Transfer Function ENTIER 2-17
Functions
Standard Functions 2-16

Get
Get and Put 3-35
Go-To
Go-To Statements 2-28

H
H Option D-3
Hardware
Hardware Representations 2-6
Hidden
Hidden Variables 3-18
Horizontal
Horizontal and Vertical Page Control 3-16

Identifiers
Identifiers or Names 2-10
Function and Procedure Identifiers 2-15
If
Conditional or If Statements 2-29
Implied
Implied For-Loop 3-29
Index
Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49
Input
Input Output 3-1
Input and Output Calls 3-22
Input Output Procedures 3-37, 3-41
Input-Output
Input-Output Errors, End-of-file End-of-Tape 3-44
Input-Output Channel Definition 7-1

Integer
Numbers Integer, Decimal, and Exponent Parts 2-10
Real and Integer Types 2-12
Interactive
Interactive Debugging under INTERCOM D-1
CHECK Command for Interactive Syntax Checking D-2
Interactive Compilation and Execution D-3
Sample Interactive Example under INTERCOM D-9
INTERCOM
Interactive Debugging under INTERCOM D-1
Sample Interactive Example under INTERCOM D-9
Interface
FORTRAN Extended and RUN Interface Macros 15-1
Interruption
'END' Interruption or Terminating Display of Data D-7

Language
Reference Manual Language Conventions and Notation 2-1
Publication Language 2-6
Reference Language Description 2-7
Letter
Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8
Linking
Linking Non-ALGOL Coded Subprograms or Procedures 15-1
List
List Procedures 3-21
Logical
Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8

Macros
COMPASS Coded Utility Macros 15-1
FORTRAN Extended and RUN Interface Macros 15-1
Messages
Compiler Diagnostics and Error Messages 8-1
Metalinguistic
Backus Metalinguistic Notation for Syntax Description 2-7

Names
Identifiers or Names 2-10
NOCHECK
NOCHECK Command D-2
Non-ALGOL
Non-ALGOL Code as Procedure Body 2-43
Linking Non-ALGOL Coded Subprograms or Procedures 15-1
Notation
Reference Manual Language Conventions and Notation 2-1
Backus Metalinguistic Notation for Syntax Description 2-7
Number
Number Formats 3-2
Numbers
Numbers Integer, Decimal, and Exponent Parts 2-10

Object-Code
Object-Code 10-1

Object-Time
Object-Time Diagnostics 8-9
Object-Time Stack 11-1
Object-Time Dump 12-1
OFF
Debugging Aids TRACE, SNAP, OFF 14-1
Operator
Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Operators
Precedence of Operators 2-23
Option
H Option D-3
Options
ALGOL Control Cards and Options 6-1
Debugging and Execution-Time Options 13-1
Output
Input Output 3-1
Input and Output Calls 3-22
Input Output Procedures 3-37, 3-41
Compiler Output 5-1
Debugging Output 14-5

Page
Horizontal and Vertical Page Control 3-16
Precedence
Precedence of Operators 2-23
Procedure
Function and Procedure Identifiers 2-15
Procedure Statements 2-23
Procedure Declarations 2-40
Non-ALGOL Code as Procedure Body 2-43
Input Output Procedures 3-37, 3-41
Linking Non-ALGOL Coded Subprograms or Procedures 15-1
Program
Compilation and Source Program Definition 4-1
Sample Program B-1
Publication
Publication Language 2-6
Put
Get and Put 3-35

Real
Real and Integer Types 2-12
Reference
Reference Language Description 2-7
Report
ALGOL-60 Revised Report 2-2
Resume
'GO' Resume Execution D-7
RUN
FORTRAN Extended and RUN Interface Macros 15-1

Sample
Sample Program B-1
Sample Interactive Example under INTERCOM D-9

Separator
Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Skipping
Skipping or Tabulating 3-35
SNAP
Debugging Aids TRACE, SNAP, OFF 14-1
Source
Compilation and Source Program Definition 4-1
Source Deck Structures 4-4
Space
Comments and Blank Space 2-9
Specificator
Delimiter Operator, Separator, Bracket, Declarator, Specificator 2-9
Stack
Object-Time Stack 11-1
Standard
Standard Functions 2-16
Standard Format 3-12
Statements
Compound Statements and Blocks 2-25
Status
'ST' Status of Active Breakpoints D-7
String
String and Alpha Formats 3-9
Strings
Character Strings 2-12
Format Strings 3-11
Structures
Source Deck Structures 4-4
Subprograms
Linking Non-ALGOL Coded Subprograms or Procedures 15-1
Subscripts
Variables, Arrays, and Subscripts 2-14
Switch
Switch Declarations 2-39
Symbols
Basic Symbols, Letter, Digit, Logical Value, Delimiter 2-8
Syntactic
Alphabetic Index of Definitions of Concepts and Syntactic Units 2-49
Syntax
Backus Metalinguistic Notation for Syntax Description 2-7
ANALYZE Command for Syntax Checking D-2
CHECK Command for Interactive Syntax Checking D-2
System
Compiler and System Description 1-1

Tab
ALGEDIT Tab Characters $ A-1
Tabulating
Skipping or Tabulating 3-35
Terminating
'END' Interruption or Terminating Display of Data D-7
Time
'CLOCK' CPU Time D-7
TRACE
Debugging Aids TRACE, SNAP, OFF 14-1
Transfer
Transfer Function ENTIER 2-17

Type
   Type Declarations 2-36
Types
   Real and Integer Types 2-12

Unit
   Unit or Channel Numbers 3-15
Utility
   COMPASS Coded Utility Macros 15-1

Variables
   Variables, Arrays, and Subscripts 2-14
   Hidden Variables 3-18
Vertical
   Horizontal and Vertical Page Control 3-16

$
   ALGEDIT Tab Characters $ D-1

'ALL'
   'ALL' D-5
'CLOCK'
   'CLOCK' CPU Time D-7
'END'
   'END' Interruption or Terminating Display of Data D-7
'FL'
   'FL' Field Length D-6
'GO'
   'GO' Resume Execution D-7
'LIST'
   'LIST' Breakpoint Options D-6
'SET'
   'SET' Breakpoints D-6
'ST'
   'ST' Status of Active Breakpoints D-7
'TIME'
   'TIME' and Date D-6

1974 Funky Winkerbean cartoon: The computer is having problems with a student's class schedule, and the administrator's solution is to expel the student. Don't ask me why I pasted this here.

TITLE: ALGOL 60 Version 3 Reference Manual

PUBLICATION NO. 60329000 REVISION D

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments:

FROM NAME: _____ POSITION: _____

COMPANY NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

CUT ON THIS LINE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

FOLD

FOLD