CONTROL DATA
CORPORATION

CONTROL DATA®
CYBER 70 COMPUTER SYSTEMS
MODELS 72, 73, 74, 76
7600 COMPUTER SYSTEM
6000 COMPUTER SYSTEMS

ALGOL REFERENCE MANUAL

CYBER 70 SERIES VERSION 4

6000 SERIES VERSION 4

7600 SERIES VERSION 4

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.
(8-15-73)	
Publication No. 60384700	

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

PREFACE

This manual describes the ALGOL-60 language (Version 4.0) for the CONTROL DATA® CYBER 70/Models 72, 73, 74 or 76; the 6000 Series computers; and the 7600 computer. It is assumed that the reader has knowledge of an existing ALGOL language and the CONTROL DATA CYBER 70 and 6000 Series computer systems.

The compiler operates under control of the SCOPE 2.0, 2.1, and 3.4 operating systems. It utilizes SCOPE multiprogramming features to provide compilation and execution within a single job operation.

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

	Publication No.
SCOPE 2 Reference Manual	60342600
SCOPE 3.4 Reference Manual	60307200
LOADER Reference Manual	60344200
COMPASS 2 Reference Manual	60279900
COMPASS 3 Reference Manual	60360900
Record Manager	60307300

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

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.

The ALGOL programming language and a compiler for translating ALGOL programs into machine language for execution by the CONTROL DATA CYBER series computers are described. 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 also in the International Organization for Standardization's Draft ISO Recommendation No. 1538, which includes the input-output definitions. Control Data's Programming Standard 1.86.003 has been used to clarify the aforementioned documents.

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 is in standard type. The name ALGOL means CYBER ALGOL, unless otherwise specified.

ALGOL SYSTEM DESCRIPTION 1

1.1 COMPILER FEATURES

The ALGOL compiler for the CYBER series computers is originally based in design on the ALGOL compiler developed by Regnecentralen, Copenhagen, Denmark, for the GIER computer. This design was adopted and, to a great degree, modified and 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.

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 overlay or non-overlay form.

Optimization facilities.

The use of a Large Core Memory and Extended Core Storage for the storage of arrays.

SOURCE INPUT

Source input is normally a card deck. 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.

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 object code is in standard relocatable binary format.

OPTIMIZATION FACILITIES

The user has the option of requesting optimization of his program at both the source language and machine code levels.

OBJECT PROGRAM EXECUTION

Execution of the object program is controlled by the Run-Time System, which is external to the generated program.

OBJECT ERROR DETECTION

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

1.2 COMPILER PACKAGE

The ALGOL compiler package consists of the following subprograms recorded on the system library:

> The compiler: ALGOL, ALGO, ALG1, ALG2, ALG3, ALG4, ALG5, and ALG6.
>
> The library subprograms which are available to object programs generated by the compiler.

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.

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

ALG1 through ALG6 each form one overlay of the compiler. Each subprogram overlays the previous one in a separate load. Each overlay generates an intermediate form of the source text which is used as input to the next overlay.

> ALG1 performs syntactic analysis of source text. If any fatal errors have been encountered, compilation terminates after this pass, providing the F option has been selected.
>
> ALG2 performs semantic analysis of the source text. If any fatal errors have been encountered in this pass or in ALG1, compilation terminates after this overlay.

ALG3 performs actual/formal procedure parameter checking and starts source language optimization. For any non-fatal errors a warning message is printed. There are no fatal errors detected by this pass. If no optimization has been requested, this overlay is not called and compilation proceeds directly from ALG2 to ALG4.

ALG4 completes source language optimization and performs the source text translation in a form suitable for code generation in ALG5.

ALG5 produces final output from the compiler, such as the object code in standard relocatable binary format.

ALG6 produces the cross-reference map and creates the dumpfile to be used at execution time by the dump routine. This overlay is called only if the R or D option is selected.

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 3.4 or SCOPE 2.0 operating system. Compilation is requested by a SCOPE control card specifying the name ALGOL. This call results in the loading and execution of the subprogram ALGOL which controls the compilation process. The compiler obtains the control card parameters from SCOPE.

1.6 MACHINE CONFIGURATION

The CDC CYBER 70 ALGOL 4.0 compiler will operate under the minimum (and maximum) basic machine configuration required by the SCOPE 3.4 and SCOPE 2.0 operating systems.

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, Appendix C.

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 procedures 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.

Material describing Control Data modifications is shaded in this manner.

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 syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic 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

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 agreement.

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 ACM 1, 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 publication, 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

```
<ab> ::= ( | [ | <ab> ( | <ab><d>
```

†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.

Sequences of characters enclosed in the brackets < > represent matalinguistic 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 recursive rule for the formation of values of the variable < ab >. It indicates that < ab > may have the value (or [or 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. Basic Symbols, Identifiers, Numbers, and Strings, Basic Concepts.

Other available characters which are not used in the hardware representation are defined to be extra basic symbols. They may occur only within strings and comment sequences.

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

†Shaded material describes Control Data modifications to the original language as presented in this report.

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.

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

†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.

‡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.

< 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.

2.3 Delimiters

The symbols code, algol and fortran defined below are added to the language to permit reference to separately compiled procedures (Section 5.4.6)

< code procedure body indicator > ::= code | algol | fortran

2.3.1 Additional Delimiters

Addition delimiters are added not to the language, but in comments. Under the normal mode of compilation, such delimiters are ignored by the compiler and thus have no effect during the execution of the resultant object code. Since these delimiters and their attached directives are acceptable ALGOL-60 character sequences, it is unnecessary to remove them a source program submitted to an ALGOL compiler in which they are not recognizable. However, if the C option is selected (Chapter 6), those additional delimiters are detected by the compiler and cause the corresponding actions to be taken for the object program. The virtual array directive, however, is always detected by the compiler and is not dependent on the C option.

Those additional delimiters are:

a) for the debugging facility – trace, snap, snapoff (Chapter 9)
b) for the overlay directive – overlay (Chapter 10)
c) for the virtual array directive – virtual (Chapter 6 – S option, and Chapter 11)
d) array bound checking directives checkon and checkoff (Chapter 6)

2.4 IDENTIFIERS

2.4.1 SYNTAX

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

2.4.1 SYNTAX

External identifier is added to the language and is defined as follows:

< external identifier > ::= < identifier with length less than or equal to 7 letters and/or digits >

2.4.2 EXAMPLES

q
Soup
V 17a
a34kTMNs
MARILYN

2.4.3 SEMANTICS

Identifiers have no inherent meaning, but serve to 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

63 is the maximum number of basic symbols of an identifier which are used for identification. Identifiers which differ in the first 63 characters are distinguished while those that do not differ are not distinguished.

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 4000 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.

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 fifteenth 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 $n < i+14$ or is d_{i+14}.

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

$$e=e_1e_2 \ldots e_m - (n-j) + \max(0,n-(i+14)).$$

If a real or integer number is larger than the largest number acceptable to the compiler, it is replaced by the largest number (cf. Section 5.1.3). In the case of integer numbers, the largest number is stored as real by appending .0 at the end and converting to floating-point representation.

If the absolute value of a real number is smaller than the absolute value of the smallest real number distinguishable from zero (cf. section 5.1.3), the compiler replaces it by 0.0. In all cases, a warning message is given.

The values of the largest number acceptable and the smallest real number distinguishable from zero are available via the standard procedure THRESHOLD (cf. Chapter 3, Section 3.5).

Real numbers are evaluated to full accuracy before rounding.

While integers and numbers are defined in this section, only unsigned integers and unsigned numbers are used elsewhere in the reference document and are recognized by the compiler. Signed numbers are treated as expressions.

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 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).

Signed numbers are treated as expressions

During object code generation, the type and value of a number may be changed depending on the operation with which it is associated in the source program (Section 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 >’

2.6.1 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’ >

The syntax of < open string > is replaced by:

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

2.6.2 EXAMPLES

‘5k,,-‘ [[[‘∧ = / : ’Tt’’
‘. .This⊔is⊔a⊔‘string’’

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 allowable characters (not basic symbols, as defined in the Report). These quote 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. Expressions

Additional constituents of expressions are labels and switch designators.

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(n×pi/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 The subscript expressions are evaluated in sequence from left to right.

An optional check is available for each subscript expression (see C option, Chapter 6, Section 6.1.2). If an attempt is made to access a subscripted variable with a subscript expression outside the corresponding subscript bounds of the array, an error is signalled if the check is operational but the effect is undefined if the check is inhibited.

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 value of a function designator.

3.2.3 Semantics

There is no restriction on side-effects, e.g., on the alteration of the values of non-local variables during the evaluation of a function designator.

It is always permissible to leave the body of a procedure which defines a function designator via a goto statement. In such a case, execution of the statement containing the function designator is discontinued.

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 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 Procedures

The list of reserved identifiers is expanded to include the following:

INLIST	GET	SYSPARAM	BACKSPACE
OUTLIST	PUT	EQUIV	DUMP
INPUT	GETITEM	STRINGELEMENT	CLOCK
OUTPUT	PUTITEM	CHLENGTH	READECS
INREAL	FETCHITEM	POSITION	WRITEECS
OUTREAL	STOREITEM	THRESHOLD	MOVE
INARRAY	H LIM	INRANGE	MATMULT
OUTARRAY	V LIM	IOLTH	VINIT
INCHARACTER	H END	ERROR	VXMIT
OUTCHARACTER	V END	CHANERROR	VMONOD
GETLIST	NODATA	ARTHOFLW	VDIAD
PUTLIST	FORMAT	PARITY	VSDIAD
FETCHLIST	TABULATION	EOF	VPROSUM
STORELIST		BAD DATA	VDOT
GETARRAY		SKIPF	BNOT
PUTARRAY		SKIPB	VBOOL
		ENDFILE	VSBOOL
		REWIND	VREL
		UNLOAD	VSREL

These procedures are described in the appropriate sections.

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 SCOPE library, any object program (Chapter 5) can call them without loading special libraries.

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> ::= X | / | ÷

<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:

$7.394_{10}{-8}$
sum
w [i+2,8]
cos (y+z×3)
(a−3/y+vu↑8)

Factors:

omega
sum ↑cos(y+z×3)
$7.394_{10}{-8}$ ↑w[i+2,8] ↑ (a−3/y+vu↑8)

Terms:

U
omega×sum↑cos(y+z×3)/$7.349_{10}{-8}$↑w[i+2,8]↑ (a-3/y+vu ↑ 8)

Simple arithmetic expression:

U−Yu+omega×sum↑cos(y+z×3)/$7.349_{10}{-8}$↑w[i+2,8]↑
(a−3/y+vu↑8)

Arithmetic expressions:

w×u−Q(S+Cu)↑2
<u>if</u> q > 0 <u>then</u> S+3×Q/A <u>else</u> 2×S+3×q
<u>if</u> a < 0 <u>then</u> U+V <u>else</u> <u>if</u> a×b> 17 <u>then</u> U/V <u>else</u> <u>if</u> k≠y <u>then</u> V/U <u>else</u> 0
a×sin(omega×t)
$0.57_{10}12$×a[N×(N−1)/2,0]
(A×arctan(y)+Z)↑(7+Q)
<u>if</u> q <u>then</u> n−1 <u>else</u> n
<u>if</u> a <0 <u>then</u> A/B <u>else</u> <u>if</u> b = 0 <u>then</u> B/A <u>else</u> z

3.3.3 SEMANTICS

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expression 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 designation 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

The primaries within a simple arithmetic expression are evaluated in sequence from left to right.

In the evaluation of the more general arithmetic expressions which include if clauses, one of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4). Such an arithmetic expression has the form: if B then E else F, where B is a Boolean expression, E is a simple arithmetic expression and F is an arithmetic expression. This selection is made as follows:

The Boolean expression B is evaluated. If its value is true, the expression E is selected. If the value of B is false, the expression F is evaluated. This expression F may of course be another arithmetic expression of this form to be evaluated according to the same rule. The type of the expression is integer if E and F are both type integer; otherwise it is real.

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 X 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/bX7/(p-q)Xv/s

means

$((((a\times(b^{-1}))\times7)\times((p-q)^{-1}))\times v)\times(s^{-1})$

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÷b=sign(a/b)×entier(abs(a/b))

(cf. sections 3.2.4 and 3.2.5).

3.3.4.2 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↑n↑k means $(2^n)^k$

while

2↑(n↑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.

$a\uparrow r$ If i > 0, exp(rXln(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 $a\uparrow i$ or $a\uparrow r$ is the same as defined above except when a is of type integer and i is a positive integer. In this case, if i is signed, then the result is of type real while if i is unsigned the result is of type integer. The Revised Report calls for a result of type integer in both cases.

If the result is specified above as undefined, an error message is issued.

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: X/÷
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), $\wedge$ (and), $\vee$ (or), $\supset$ (implies), and $\equiv$ (equivalent), is given by the following function table.

b1	false	false	true	true
b2	false	true	false	true
¬b1	true	true	false	false
b1$\wedge$b2	false	false	false	true
b1$\vee$b2	false	true	true	true
b1$\supset$b2	true	true	false	true
b1$\equiv$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: $< \leqslant = \geqslant > \neq$

third: ¬

fourth: $\wedge$

fifth: $\vee$

sixth: $\supset$

seventh: $\equiv$

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. Therefore, the definition of < label > is replaced by:

< label > ::= < identifier >

Note also that the first example in 3.5.2 is incorrect.

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 syntactic structure, the syntactic 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 statement 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
```

4.1.3 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 non-local 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

Program labels are not allowed in ALGOL 4.0. The maximum number of blocks permitted in a single ALGOL 4.0 compilation is 253. The maximum static nesting permitted for blocks is 63. A procedure body counts as a block in this context. Blocks may be overlayed; for a description of this facility, see Chapter 10.

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-q×S
S [v,k+2] : = 3-arctan(s×zeta)
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 Semantics

Assignment to a procedure identifier may occur only within the body of a procedure defining a function designator of the same name.

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

4.2.3.1 This section is ambiguous since subscripts may contain subscripted variables. It is amended to:

The subscript expressions of any subscripted variables occurring as left part variables are evaluated in sequence from left to right.

If the assignment statement is in a procedure body, any formal parameters in the left part list are replaced by actual parameters according to section 4.7.3.2. before this evaluation of subscript expressions takes place.

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.

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.

If the designational expression of a go to statement is a switch designator whose value is undefined when an attempt is made to execute the go to statement, the object program terminates abnormally.

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 := 2×a 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

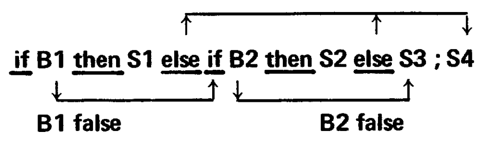
else < unconditional statement >

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:



4.5.3.2 Conditional statement

According to the syntax, three forms of unlabelled conditional statements are possible.

These may be illustrated as follows:

if B then S
if B then S else T
if B then U

Here B is a Boolean expression, S is an unconditional statement, T is a statement, and U is a for statement.

The execution of a conditional statement may be described as follows:

The Boolean expression B is evaluated. If its value is true, the statement S or U is executed. If its value is false and if the conditional statement has the second form, the statement T is executed (this statement may of course be another conditional statement, to be interpreted according to the same rule).

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.

4.5.4 Go to into a conditional statement

If a go to statement refers to a label within S or U, the effect is the same as if the remainder of the conditional statement ('if B then' and in the second case also 'else T') were not present.

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:

```
             ┌— — — — — — — ┐
             ↓               ↑
Initialize ; test ; statement S ; advance ; successor
             ↓                           ↑
             | for list exhausted        |
             — — — — — — — — — — — — — —
```

In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test 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;
      statement  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.

4.6.4.2 Step-until-element. The execution governed by a step-until element is as described so far as possible side-effects are concerned. However, determination of V is performed only once, at the entry to the for clause. This can result in side-effects in the case where V is a subscripted variable containing a call to a procedure which modifies the value of V.

Secondly, the step element is evaluated only once for each execution of the step expression. That value is used to increment the for variable and to multiply with when testing the limit.

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 ;
          Statements S ;
          go to L3 ;
```

where the notation is the same as in 4.6.4.2. above.

4.6.4.3 While-element. The execution governed by a while element is as described so far as possible side-effects are concerned. However, the determination of V is performed only once, at the entry to the for clause. This can result in side-effects in the case where V is a subscripted variable containing a call to a procedure which modifies the value of V.

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.

4.6.5 The value of the controlled variable upon exit

In the case of step-until and while-elements, the value of the controlled variable on exit from a for statement through exhaustion of the for list is the value causing the failure of the test. In the case of a simple for element, the value of the controlled variable on exit is the same as it was at the end of the final execution of the statement S.

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

Any occurrence of a label in a designational expression outside the for statement which contains that label and the statement it labels will cause a warning message at compile time. Any go to statement outside a for statement referring to a label within the for statement will thus cause this warning. However, the legitimate case of a go to statement in a procedure called from within a for statement which refers to a label within that for statement via an actual parameter of the procedure call will not cause a warning. It is permissible to specify a transfer from within a procedure to a label in the block calling the procedure.

The effect of executing a go to statement of this kind is undefined despite the warning.

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 non local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.1 Value assignment (call by value)

The actual parameters corresponding to formal parameters called by value are first evaluated from left to right for non-arrays, then from left to right for arrays, in sequence as they appear in the actual parameter list.

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 conficts 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 in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is 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 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.

A label cannot be specified by value. The types of real and integer actual parameters should correspond to the types of the formal parameters. However, the control card option X (Chapter 6, section 6.1.2) permits mixed actual-formal correspondence for reals and integers at the possible expense of object time efficiency.

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 symbols code, algol and fortran are included to permit reference to procedures which are compiled separately from the program of 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 the 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 >

The difference between own and non-own quantities is the following:

Non-own variables are attached to the block in which they are local both with regard to the meaning of their identifiers and with regard to the existence of their values. Each entry into a block, whether recursive or not, will bring a new set of the non-own quantities of that block into existence, the values of which are initially undefined. On exit from a block all the non-own quantities created at the corresponding entry are lost, with respect both to their identifiers and to their values.

Own quantities, on the other hand, are local only with respect to the accessibility of their identifiers. Where the existence of their values is concerned they behave more closely as though declared in the outermost block of the program. Thus every entry into a block, whether recursive or not, will make the same set of values of the own quantities accessible. On exit from a block the values of the own quantities of the block are preserved, but remain inaccessible until re-entry is made to a place within the scope of the identifiers.

Each own variable and each subscripted variable corresponding to an own array is initialized to one of the following values, according to type (i.e., it is given that value on its creation by entry into the block in which it is declared):

Type	Initial Values
integer	0
real	0.0
Boolean	false

Certain identifiers have meaning without explicit declaration. These are the standard procedures of sections 3.2.4 and 3.2.5 and behave as though they were declared in a block embracing the entire program (including the fictitious block supplied for labels in section 4.1.3) and any implicit outer blocks. All other identifiers of a program must be declared.

Further such identifiers can be added by implicit outer blocks (see Chapter 4).

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 value 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. All numbers are normalized and the complete range of absolute values is

$$(2\uparrow 47)*2\uparrow(-1022) \leq \text{number} \leq (2\uparrow 48-1)*2\uparrow 1022$$

which is approximately

$$1.6_{10}{-294} \leq \text{number} \leq 1.2_{10}322$$

The largest integer, N, which can be represented in ALGOL 4.0 such that the representation of N-1 ≠ N and the representation of N+1 = N is

$$2\uparrow 48 = 281\ 474\ 976\ 710\ 656$$

The smallest number, P, which can be distinguished when added to 1.0 is $2\uparrow{-47}$ which is approximately $3.6_{10}{-15}$.

Real and integer values with up to 15 significant decimal digits can be represented.

A real or integer zero is represented by a word of 60 zero bits.

Standard procedures THRESHOLD and INRANGE are provided for setting and testing limiting numeric values. (Chapter 3, section 3.5.)

The machine provides representation for both ± ∞ and ± indefinite values and the use of such values as operands in arithmetic operations produces run-time error conditions called mode errors.

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

The values of own variables are global to the whole program because of their assigned position in the object program stack. They are however, accessible only in the block in which they are declared in the same way as any other declared variable.

5.2 ARRAY DECLARATIONS

5.2.1 SYNTAX

< lower bound > ::= < arithmetic expression >

< upper bound > ::= < arithmetic expression >

< bound pair > ::= < lower bound > : < upper bound >

< bound pair list > ::= < bound pair > | < bound pair list > , < bound pair >

< array segment > ::= < array identifier > [< bound pair list >]|< array identifier > , < array segment >

< array list > ::= < array segment > | < array list > , < array segment >

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

5.2.2 EXAMPLES

```
array a,b,c [7:n,2:m] ,s [-2: 10]
own integer array A [if c < 0 then 2 else 1 :20]
real array q [-7:-1]
```

5.2.3 SEMANTICS

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

5.2.3.1 Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2 Dimensions. The dimensions are given as the number of entries in the bound pair lists.

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

5.2.3 Semantics

Normally, an array is stored in CM or SCM. The user can allocate an array to ECS or LCM with the S option. The user may declare an array to be virtual. A virtual array can be referenced only in entirety and can be used only as a parameter to a procedure. (See Chapters 6 and 11, and the MOVE procedure in section 3.5, Chapter 3).

The bound pairs in a bound pair list are evaluated in sequence from left to right. In each bound pair, the lower bound is evaluated before the upper bound.

Array bound checking for each index of an array is performed for an array specified in the comment directive, checkon (see the C option in Chapter 6).

The maximum number of dimensions permitted in the declaration of an array is 63.

own arrays with dynamic bounds (bounds which are not constants in the program) are not permitted. Thus the following declaration, given as an example in Section 5.2.2 is illegal:

```
        own integer array A [ if C < 0 then 2 else 1 :20]
```

5.2.4 LOWER UPPER BOUND EXPRESSIONS

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2.).

5.2.4.2 The expressions can only depend on variables and procedures which are non local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3 An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4 The expressions will be evaluated once at each entrance into the block.

5.2.5 THE IDENTITY OF SUBSCRIPTED VARIABLES

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared own the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3 SWITCH DECLARATIONS

5.3.1 SYNTAX

< switch list >::= < designational expression > | < switch list > < designational expression >

< switch declaration > ::= switch < switch identifier > := < switch list >

5.3.2 EXAMPLES

```
switch S := S1,S2,Q[m] ,if v > -5 then S3 else S4
switch Q:= p1,w
```

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 >

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 >

  < external identifier > ::= < identifier with length less than or equal to 7 letters and/or digits >

  < code identifier > ::= < code number > | < external identifier > | < empty >

  < code > ::= code < code identifier > | algol < code identifier > |

    fortran < code identifier >

For the semantics of code see section 5.4.6.

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 n 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
```

5.4.3 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. An identifier may not occur more than once in a formal parameter list.

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 the 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.

5.4.4 Values of function designators

Since function designators may occur in subscripts, this section should read ". . .as a left part. . ." rather than ". . .in a left part". This appears twice.

5.4.5 SPECIFICATIONS

In the heading of 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, whether called by name or by value, 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 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, algol, or fortran followed by a code identifier. This may be either a number XXXXX in the range 0-99999, or name of seven or fewer characters, or empty. If the name has more than 7 letters or digits, only the first 7 are used. In the case where the code identifier is empty, the name of the procedure in which it is declared, will be assumed to be the name of the external identifier.

The code identifier is either the same number or the same external identifier associated with the procedure when it is compiled separately as an ALGOL or FORTRAN Extended source procedure (Chapter 4) and the compiler uses this number or this external identifier to link the declaration with the procedure. The procedure is linked to the main program at the object program loading. In the case of a procedure which utilizes the symbol code or algol, 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 the handling of formal parameters conforms to the object code produced by the compilation of an ALGOL source procedure. The macros of ALGTEST may be used for this purpose (Appendix B).

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 or by external identifier. However, all references in the program must use the name declared in the program rather than the name declared in the separately declared procedure. Please note that in the case where the code identifier is empty, the name in the procedure heading is taken to be the external identifier and hence must agree with the name of the separately declared procedure.

Only simple variables, constants, and arrays may be passed as FORTRAN parameters.

To define an external source procedure, the source procedure has the form of a procedure declaration preceded by:

```
    code < code number >; or code < external identifier >;
    algol < code number >; or algol < external identifier >;
```

To call an external source procedure, the calling source procedure or source program must contain a declaration for that procedure of the form:

```
    procedure < procedure heading > < code >
    procedure < code procedure heading > < code >
    < type > procedure < procedure heading > < code >
    < type > procedure < code procedure heading > < code >
```

where < code procedure heading > ::= < procedure identifier > < formal parameter part >

The declaration may, but not necessarily, contain specifications of all formal parameters of the procedure. These specifications will have no effect.

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 the compilation of AVERAGE is:

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

followed by the 'EOP.'

The second procedure body is replaced by the symbol code and the identifying number 527. The procedure heading remains the same. 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);
            value LOW, HIGH;
            real 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
```

The following example depicts a source procedure with the external identifier FAC:

```
    algol FAC ; integer procedure FACTORIAL (n) ;
          value n ; integer n ;
          FACTORIAL : = if n = 0 then 1
                          else n X FACTORIAL (n-1) ;
```

Note that the external identifier and procedure identifier may be the same.

The above external procedure may be declared by the following declaration:

```
    integer procedure ABC (n) ; value n ; integer n ; algol FAC ;
```

Note that all references to the procedure within the source program or source procedure would be to ABC.

The facility offered by the external name allows a library of object program procedures to exist which were compiled with arbitrary identifiers but distinguishing external names.

The names of the formal parameters in the procedure heading need not be the same as those declared when the procedure is compiled separately, but the number of parameters must be the same.

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.

The above external procedure could also have been declared by the following declaration:

```
    integer procedure FAC (n) ; value n ; integer n ; algol ;
```

Since the code identifier following algol is empty, FAC, the name declared in the procedure heading, is used.

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 suitable 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 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: =j+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 ;
```

comment: RK integrates the system Yk' = fk(x,$y_1 y_2$,. . . ,y_n) (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 $y_k(x)$. 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 f_k. 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

[†]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, Fërhd. 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.

always be given the value true for an isolated or first entry into RK. If however the function y must be available at several meshpoints $x_0,x_1, \ldots ,x_n$, then the procedure must be called repeatedly (with $x=x_k$, $xE=x_{k+1}$, 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 $z[k] = f_k(x,y[1],y[2], \ldots ,y[n])$ 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
parameters 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,b,c,) 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.

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 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. Seegmüller, 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.

[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.

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

A.1.1 Syntax

Basic components:

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

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

< 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. . .	.1008$_{10}$+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. When a format string is contained in an array allocated in LCM, the array is called by value by any library input/output procedure, otherwise by name.

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 equivalent to 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).

†Shaded material represents modifications to the original ALGOL report.

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 262,143.

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.1.3.2 Insertions. Note that insertions may include any of the basic 6-bit display BCD characters according to string definition (cf. Chapter 2, section 2.6.1).

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.5). 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} entier $(10^{d} X + 0.5)$
Truncation	10^{-d} sign (X) entier $(10^{d}$ abs (X))

A.1.3.5 Truncation. On output, the number of significant digits appearing for a real number cannot exceed the degree of accuracy available in 60-bit floating point form. Thus at most 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 has 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.5).

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 position 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 an error is found in the data, no attempt is made to reread the data as if it were in standard format. An error message is given.

If a number is read which is larger than the largest number acceptable and its destination is real, then the destination is set to the largest number (cf. Chapter 2, Section 5.1.3) with the appropriate sign. If a number is read which is smaller than the smallest real number distinguishable from zero and its destination is real, then the destination is set to 0.0. If a number larger than the largest number acceptable is read and its destination is integer, an error occurs.

The values of the largest number acceptable and the smallest real number distinguishable from zero are available via the standard procedure THRESHOLD (cf. Chapter 3, Section 3.5).

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>

A standard format item, N, has been added.

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

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. 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.2.3.1 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. 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 characters in the string S. If the string S contains less than 8 characters, then zeroes (empty characters) are added to the right. Thus the value of EQUIV(S) is the same as if the string character had been input under A format. The EQUIV function, and input under A format assign an internal representation of zero (an empty character) to blank characters, thus some limited string manipulation using integer variables can be achieved.

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 behaviour 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 octal 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. This can be used to input format strings (cf. Section B.3.1).

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 or 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 representation 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 indicate 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) ,//’
    ‘↑’
    ‘.5D₁₀+D,X (2 (20B.8D₁₀+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 262,143 repetitions. The compiler does check the syntax of strings (see section 3.9 on the efficient use of this facility).

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;
*	page alignment

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 $+D.14D_{10}+3D$. When the number cannot be written using the given format, the modified standard format is $`*'+D.14D_{10}+3D`*'$ (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 none is 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 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.DDDDDD$_{10}$+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,‘↑’);

<u>for</u> i := 1 <u>step</u> 1 <u>until</u> n <u>do</u>

```
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('1,(X(BB-ZZZZ.DD),//)',n)
```

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

The format string is 1.

(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 := 1 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.

In order to be compiled without diagnostic the above example must be written with specifications (Chapter 2, Section 5.4.5).

```
procedure LIST(ITEM);procedure ITEM;

    for i := 1 step 1 until n do

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

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 – 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

Each channel is designated by a unique non-negative integer. If channel 0 is used, a channel card defining it as indexed must be supplied. Channels are established by means of channel cards (Chapter 7).

Channels are divided in four mutually incompatible sets according to the procedures that are available to transfer data:

1. Coded Sequential – INCHARACTER, OUTCHARACTER, INREAL, OUTREAL, INARRAY, OUTARRAY, INPUT, OUTPUT, INLIST, OUTLIST
2. Indexed – GETLIST, PUTLIST, GET, PUT, GETITEM, PUTITEM
3. Word Addressable – FETCHLIST, STORELIST, STOREITEM, FETCHITEM
4. Binary Sequential – GETARRAY, PUTARRAY.

Any attempt to use a channel as a member of more than one set within a program will cause an error.

Input-output procedures for direct access devices (indexed and word addressable sets) will be described in section 3.2.2 and those for binary sequential channels in section 3.2.3. Serial channel characteristics and procedures are described below.

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 position L through R of a horizontal line are used, and only lines L′ and R′ of the page are used; we require that $1 \leqslant L \leqslant R$ and $1 \leqslant L' \leqslant R'$. 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 $P \geqslant R$ and $P' \geqslant R'$, 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 $L \leqslant P$ and $L' \leqslant P'$ 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).

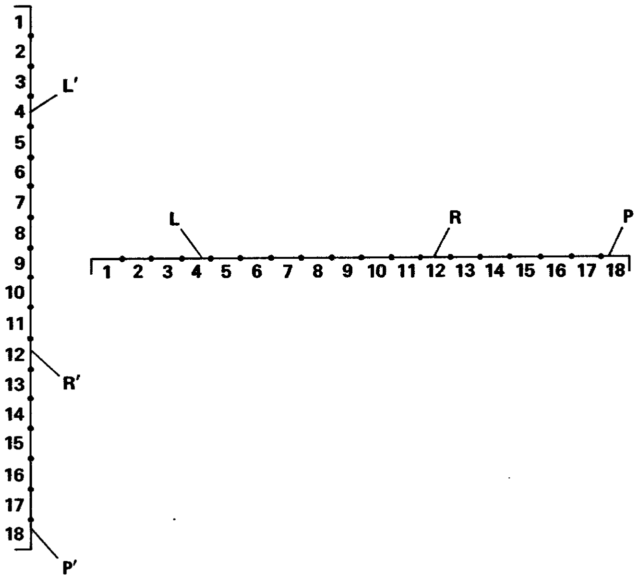


Figure 2.

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 $P' = \infty$. 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 $P = P' = \infty$. 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 value H6 (Section B.3.3). Note that the hidden variable corresponding to NO DATA is H7 rather than H6.

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.

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

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

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, . . .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 . 10D₁₀+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.

The number of X replicators is limited only by the fact that there may be a maximum of 63 actual parameters in a procedure call.

If the format string in a format call contains nX replicators, it must be followed by at least n non-negative integer parameters. If the number, n, of non-negative integer parameter exceeds the number of X replicators, then the n X replicators in the format string will be replaced by the first n parameters; however, if there are less integer parameters than required by the format string, an error occurs.

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 $R = R' = \infty$.

B.3.2 Limits

Any attempt to set $R \leqslant L$ or $R' \leqslant L'$ will be ignored. If $L > P$, the value 1 is substituted for L; similarly, if $L' > P'$, the value 1 is substituted for L'.

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 H6 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 H6 = 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 labelled L will occur. If the procedure no data is 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

The call to NO DATA sets the hidden variable H7 rather than H6.

End of data is defined as the occurrence of an end-of-file or end-of-record 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 or CHANERROR procedure (Section 3.3). If neither of these procedures was called or if an 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	$(1,\infty)$	$(1,\infty)$
H4	h end	(, ,)	(, ,)
H5	v end	(, ,)	(, ,)
H6	tabulation	1	1
H7	no data	L32	end program

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.4.1 General characteristics

List procedures are used in conjunction with the list-driven I/O procedures to describe the sequence of items which is to be transmitted for intput or output. A list procedure has one parameter which is itself a procedure, for example named ITEM. Each I/O item is written as a parameter to procedure ITEM, e.g., ITEM (u). When the list procedure is called by an I/O system procedure, the internal system procedure INITEM or OUTITEM is effectively 'substituted' for ITEM, u is called by name and the value of u is transmitted for intput or output.

In order to be compiled without diagnostic the above 2 examples must be written with specifications (Chapter 2, Section 5.4.5).

```
    procedure LIST (ITEM); procedure ITEM;

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

    procedure PAIRS (ELT); procedure ELT;

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

            ELT (B[i]) end
```

B.4.1.1 Additional parameter type

An array name may be a parameter to the following procedures: OUTPUT, INPUT, GETITEM, PUTITEM, FETCHITEM, STOREITEM. A call of these procedures with an array parameter is equivalent to a sequence of calls with each of the subscripted variables of the array as parameters in lexicographical order, i.e., with last subscript varying fastest. Note that each subscripted variable uses one format item.

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:

INCHARACTER	INREAL	INARRAY
OUTCHARACTER	OUTREAL	OUTARRAY
CHLENGTH		

Character Transmission

The procedures INCHARACTER and OUTCHARACTER provide the means of communicating between input-output devices and the variables of the program in terms of basic characters. (The basic characters are given in Table 1, Appendix C.) The procedure calls are as follows:

IN CHARACTER (channel, string, destination)

OUT CHARACTER (channel, string, source)

Where channel and source must be arithmetic expressions called by value;

string is a string, called by name (Chapter 2, Section 2.6.1);

destination is an integer variable called by name.

The corresponding procedure declarations may be defined as:

```
procedure  INCHARACTER (channel, string, destination); value channel;
           integer channel, destination; string string;
           < procedure body >

procedure  OUTCHARACTER (channel, string, source); value channel, source; integer channel, source; string string;
           < procedure body >
```

In both procedures, the correspondence between the basic character and the value of the variable in the program is established by mapping the sequence of basic characters given in the string (second parameter), taken from left to right, onto the positive integers 1,2,3, . . . Using this correspondence, the procedure INCHARACTER will assign to the type integer variable given as the third parameter the value corresponding to the next basic character appearing on the input stream. If this next basic character does not appear in parameter string, the number 0 will be assigned. If the next character appearing in the input stream is not a basic character, a negative integer corresponding to the symbol will be assigned.

For example, the procedure call INCHARACTER (61, 'ABC', i) will set i = 1 if the next character on the input device is A, i = 2 if B, i = 3 if C, i = 0 if a basic character other than A, B and C, i = corresponding negative integer if a non-basic character.

Similarly, the procedure OUTCHARACTER will transfer the basic character corresponding to the third parameter to the output device. If the value of this third parameter is negative a character corresponding to this value will be transferred. It is understood that where the device may be used for both INCHARACTER and OUTCHARACTER, the negative integer value associated with each additional character will be the same for the two procedures.

The following negative number is assigned to the non-basic ALGOL character

    End of line symbol : -1

There is an integer procedure, CHLENGTH, used to calculate the length of a given (actual or formal) string. The procedure call is as follows:

    CHLENGTH (string)

and the procedure declaration is:

    <u>integer</u> <u>procedure</u> CHLENGTH (string); <u>string</u> string; < procedure body >

The value of CHLENGTH (string) is equal to the number of basic characters of the open string enclosed between the outermost string quotes.

<u>Type Real Transmission</u>

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 called by value and destination is a variable of type <u>real</u> called by name, as follows:

```
procedure  INREAL (channel, destination); value channel; integer channel;
           real destination;
           < procedure body >
```

    procedure OUTREAL (channel, source); value channel, source; integer channel;
              real source;
              < procedure body >

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 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 (cf. Section A.5).

Array Transmission

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, as follows:

    Procedure INARRAY (channel, destination); value channel; integer channel;
              array destination;
              < procedure body >

    Procedure OUTARRAY (channel, source); value channel; integer channel;
              array source;
              < procedure body >

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

That is, an array is stored by rows.

It should be recognized that the possibly multidimensional structure of the array is not reflected in the corresponding number 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.

Examples:

The following examples may help to clarify the uses of the procedures described above:

(a)
```
procedure outboolean (channel, boolean); value boolean; integer channel; Boolean boolean; comment this
procedure outputs a Boolean value as the basic symbol true or false; begin integer i;
if boolean then begin for i : = 1 step 1 until 7 do
                                       OUTCHARACTER (channel, ' ' TRUE '⊔ ' ,i)
              end

              else for i : = 1 step 1 until 7 do
                             OUTCHARACTER (channel, ' ' FALSE ' ',i)
              end
```

(b)
```
procedure outstring (channel, string); value channel; integer channel; string string; comment outputs the
string to the output device;
begin integer i;
      for i: = 1 step 1 until CHLENGTH (string)
      do OUTCHARACTER (channel, string, i)
end
```

(c)
```
procedure   integer (channel, integer); value channel; integer channel,integer; comment inputs an integer which,
on the input device, appears as a sequence of digits, possibly preceded by a sign and followed by a comma.
Any symbol in front of the sign is discarded;

begin integer n,k; Boolean b ;

      integer : = 0 ; b : = true;

      for k   : = 1, k+1 while n = 0 do INCHARACTER (channel, '0123456789–+ ',n);

      if n  = 11 then b:= false ;

      if n > 10 then n: = 1 ;

      for k   : = 1, k+1 while n ≠ 13 do

               begin integer : = 10 * integer + n – 1;

               INCHARACTER (channel, '0123456789–+,',n)

               end k ;

if ¬ b then integer : = – integer

end
```

```
(d)  begin   begin array a [1:10] ;< statements > ;

             OUTARRAY (15, a)

             end
             begin array b [0 : 1, 1 : 5] ; INARRAY (15, b);
             < statements >
             end

     end
```

(e) the following example exhibits the use of INARRAY and OUTARRAY for inversion of a matrix including transfer of the matrix elements from and to the external device. It requires that an appropriate declaration for a matrix inversion procedure as well as the declaration of outstring as given above in example b are inserted at appropriate places in the program.

```
    begin integer n; INREAL (5, n) ; comment the matrix elements must be preceded by the order;

             begin array a [1 : n , 1 : n] ; INARRAY (5, a);
             matrix inversion (n, a, singular) ; OUTARRAY (15, a) ;
             go to Ex
             end;
             singular : outstring (15, 'singular');
             Ex : end
```

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)

$n = 0, \infty$

The number of e_1 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, e1)
```

is equivalent to

```
    OUTPUT 1 (channel, format string, e1)
```

defined in the proposal.

The parameter e_i may be any of the types described in Section B.4.1.1.

For example:

```
    OUTPUT (61, '/, + 3ZD.2D',A)

    OUTPUT (61, '3 (N), 'TOTAL' ', X, Y, Z)
```

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

```
    procedure output n (unit, format string, e1,e2, . . . ,en)

    begin procedure A; format (format string);

    procedure B(P); begin P(e1); P(e2); . . . . ;P(en) end;

    out list (unit, A,B) end
```

The procedure OUTPUT is defined in terms of OUTLIST as follows:

```
    procedure OUTPUT (channel; format string, e1,e2, . . . ,en) ;

        value channel; integer channel; string string; comment e1,e2, . . . ,en may be:

              -    a string
              -    any integer, real or Boolean expression,
              -    any integer, real or Boolean array;


        begin procedure a; FORMAT (format string);

              procedure b (p); procedure p ;

                    begin integer i;
                          for i : = 1 step 1 until n do p (ei)
                    end;

              OUTLIST (channel, a, b)

        end
```

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

Let the variables p and p′ indicate the current position in the output for the unit under consideration, i.e., lines 1,2, . . . ,p′ of the current page have been completed, as well as character positions 1,2, . . . ,p of the current line (i.e., of line p′+1). At the beginning of the program, $p = p' = 0$. 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 $L = 1$, $R = \infty$

H3 is set so that $L' = 1$, $R' = \infty$

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedures 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 = <u>true</u>, 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 <u>true.</u> 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 $R \leqslant P$) 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 processes 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 $L = 1, R = \infty$.
>
> H3 is set so that $L' = 1, R' = \infty$.
>
> H4 is set so that $P_{N'}, P_{R'}, P_{P'}$ are all effectively equal to the DUMMY procedure defined as follows: "<u>procedure</u> DUMMY;;".
>
> H5 is set so that P_N, P_R, P_P are all effectively equal to DUMMY.
>
> H6 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. 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, H6, which are local to OUT ITEM, the procedure will then continue at step 4.

Step 4. (Alignment marks)

If the next format item includes an alignment mark it is removed from the format string and process A (a subroutine below) is executed for each /, process B for each ↑ and process C for each J. Note that overflow procedures may cause the format strings to be changed. In such cases, the new format string is not examined before step 6.

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. (Note that the number of characters, s, may depend upon the value being output using A, H or S format, as well as on the output medium.)

Step 7. (Check for overflow)

If $\rho + s \leqslant R$ and $\rho + s \leqslant P$, 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. Otherwise, if the present item uses A, H or S format or title sequences a special non-basic symbol is output; this is done to ensure that input will be inverse to output. Continue at step 8.

Step 8. (Processing of overflow)

Process H $(\rho + s)$ is performed. Then if $\rho + s \leqslant R$ and $\rho + s \leqslant P$, step 9 is executed; otherwise k is set to min $(R, P) - \rho$. $x_1 x_2 \ldots x_k$ are output, ρ is set = min (R, P), and $x_1 x_2 \ldots x_{s-k}$ to $x_{k+1} x_{k+2} \ldots x_s$. s is decreased by k and step 8 repeated.

Step 9. (Finish the item)

$x_1 x_2 \ldots x_s$ are 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 D is executed and procedure P_N is called. 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 $k \geqslant$ 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.

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 $1 \leqslant L' \leqslant R'$ and $1 \leqslant L' \leqslant P'$. (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 $1 \leqslant L \leqslant R$ and $1 \leqslant L \leqslant P$. (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, H6 local to OUT ITEM.

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 e parameters.

The parameter e_i may be any of the types described in Section B.4.1.1.

The procedure INPUT may be defined in terms of INLIST as follows:

```
procedure INPUT (channel, format string, e1,e2, . . . ,en);
    value channel; integer channel; string format string;
    comment e1,e2, . . . ,en may be:

        – any integer, real or Boolean variable
        – any integer, real or Boolean array;

    begin procedure a; FORMAT (format string);

        procedure b (p); procedure p ;

            begin integer i;
                for i : = 1 step 1 until n do p (ei)

            end;

        INLIST (channel, a, b)

    end
```

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 processes 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 $L = 1, R = \infty$.

H3 is set so that $L' = 1, R' = \infty$.

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 so that TAB = 1.

H7 is set to terminate the program in case the data ends.

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 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 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 an alignment mark at its left, it is removed from the format string and process A (a subroutine below) is executed for each /, process B for each ↑, and process C for each J. Note that overflow procedures may cause the format string to be changed. In such cases, the new format string is not examined before step 6.

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. If the format item is A or N, s is set to 1 and step 7 is executed; otherwise, the number of symbols, s, needed to represent the formatted item on the present medium, is determined from the format item.

Step 7. (Check for overflow)

If the present item uses N format, the character positions $\rho + 1, \rho + 2 \ldots$ 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 $\rho = \min(R, P)$. If N format is not used, step 8 is executed if $\rho + s > \min(R, P)$, or step 9 if $\rho + s \leqslant \min(R, P)$.

Step 8. (Processing of overflow)

Process H (ρ + 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.

A format: Characters are input as with N format until a basic symbol has been input. (This basic symbol may use several character positions on the input medium.)

Other: if $\rho + s \leqslant R$ and $\rho + s \leqslant P$, step 9 is executed; otherwise input $k = \min(R, P) - \rho$ characters, set $\rho = \min(R,P)$, decrease s by k, and repeat this step.

Step 9. (Finish the item)

If any format other than N and A 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 and A 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 $\rho \geqslant 0$, process D is executed and procedure P_N is called. 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 $k \geqslant \min(R, P)$, 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)

If the input file is paged, records are skipped until a record is found which contains a new page mark. Otherwise, the input file is skipped to the next line. In either case ρ and ρ' are set to 0.

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 F repeated. If $\rho' + 1 > P'$: process E is executed, $P_{P'}$ called, and process F repeated. This process must terminate because $1 \leqslant L' \leqslant R'$ and $1 \leqslant L' \leqslant P$. (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 $1 \leqslant L \leqslant R$ and $1 \leqslant L \leqslant P$. (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 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 $\rho =$ position; but if $\rho \geqslant$ 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 - ρ') times; but if $\rho' \geqslant$ position, call the procedure OVERFLOW.

B.5.3 Skipping

The procedure 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 implemented as special cases of GETLIST and PUTLIST assuming the special channel number 0 and using an index derived from the integer parameter. This index is not available to the programmer.

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 Procedures

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, 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 (further defined below) 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.

---

†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 $> \rho$, insert blanks until ρ = quantity. If quantity $\leqslant \rho$ perform a line advance operation, set $\rho = 0$ and insert blanks until ρ = quantity. When setting ρ' if quantity $> \rho'$ perform line advance operations until ρ' = quantity. If quantity $\leqslant \rho'$ skip to next page, set $\rho' = 0$ and perform line advance operation until ρ' = quantity. The action for function=2 or 4 depends on the last operation on the file. If it was: read, characters or lines are input; write, characters or lines are output, no previous operation or one that closes the file (e.g., rewind), no action takes place.

‡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.

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 tapes, 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 sections 3.2–3.5, and 3.10.

Primitive Procedures	CHLENGTH
	STRING ELEMENT
I/O Procedures for Direct Access Devices	GETLIST
	PUTLIST
	GETITEM
	PUTITEM
	GET
	PUT
	FETCHLIST
	STORELIST
	FETCHITEM
	STOREITEM
I/O Procedures for Binary Sequential Files	GETARRAY
	PUTARRAY
Control Procedures	ARTHOFLW
	PARITY
	EOF
	BAD DATA
	ERROR
	CHANERROR
Hardware Function Procedures	SKIPF
	SKIPB
	END FILE
	REWIND
	UNLOAD
	BACKSPACE

Miscellaneous Procedures	IOLTH POSITION DUMP CLOCK THRESHOLD INRANGE MOVE
Extended Core Storage Large Core Memory Procedures	READECS WRITEECS

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.

```
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 X N, 63 + 3 X N)
                                                        end;

printer : = 3;

output 0 (printer,'↑ 'PASCALS⊔TRIANGLE'//');

for N : = 0 step 1 until 19 do
          begin A[N] : = 1;
          for K : = N - 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. A., 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[†]

3.2.1 PRIMITIVE PROCEDURES

An additional set of primitive procedures exists without declaration, as follows:

```
CHLENGTH (string)
STRING ELEMENT (s1, i, s2, x)
```

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.

CHLENGTH may be defined as follows:

```
integer procedure CHLENGTH (string);
string string;
comment evaluate the number of character positions string would
require if output using S format;
< procedure body >
```

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.

STRING ELEMENT can be defined as follows:

```
procedure STRING ELEMENT (s1, i, s2, x); value i; integer i, x;
string s1, s2;
comment select the ith symbol in s1, search string s2: if a
match is found assign to x the position number of the corresponding
symbol in string s2, if no match is found assign 0 to x;
< procedure body >
```

[†]Since the remainder of this chapter deals entirely with Control Data only features, shading is not used.

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, as follows:

```
REWIND (channel);
OUTCHARACTER (channel, s1, i);
REWIND (channel);
INCHARACTER (channel, s2, x);
```

3.2.2 INPUT-OUTPUT PROCEDURES FOR DIRECT ACCESS DEVICES

3.2.2.1 INDEXED LIST INPUT AND OUTPUT

```
GETLIST (channel, index, list)
PUTLIST (channel, index, list)
```

Here channel is an integer parameter called by value and is the number associated with the input-output device. The parameter index is an integer called by value which is used to identify the list of items on the input-output device. The parameter list is the name of a list procedure (cf. Section B.4)

These 2 procedures form a pair. The effect of PUTLIST is to output to the output device the items presented by a call of the list procedure, the whole set being indexed by the index.

The effect of GETLIST is to input from the input device the items indexed by the index, and transfer the values to the items presented by a call of the list procedure.

Any item output by PUTLIST can be input to a compatible item by GETLIST where compatibility is defined as:

item type for PUTLIST		item types for GETLIST
real or integer array		real or integer array
Boolean array	compatible	Boolean array
real or integer value		real or integer variable
Boolean value		Boolean variable

If an attempt is made to use GETLIST with an incompatible item an error message is given.

In the case of arrays, the elements are taken in lexicographical order, without regard to any multi-dimensional structure. Any necessary type-conversion takes place during GETLIST according to the standard rules. If an array on the input medium is larger than the GETLIST item, superfluous elements are ignored; if smaller, superfluous elements in the item are unchanged.

If during the evaluation of an I/O item, an attempt is made to input or output to the same channel, the result is undefined, e.g.:

```
procedure A (x); procedure x;
begin GETITEM (3, 72, B); x (B);
end;
PUTLIST (3, 64, A)
```

In this example, the call to GETITEM will be in error and a message will be issued.

The items provided by the list procedure can be arrays or arithmetic or Boolean expressions. (See Section B.4.1.1.)

An index is a non-negative parameter used to identify the location of the items on the external device. If more than one list of items with the same index is output to an indexed file and an attempt is made to input a list with that index from that file, then the effect is undefined.

3.2.2.2 INDEXED ITEM INPUT AND OUTPUT

GETITEM (channel, index, e_1, e_2, ... ,e_n)
PUTITEM (channel, index, e_1, e_2, ... ,e_n)

These procedures may be defined in terms of GETLIST and PUTLIST as follows:

```
procedure  GETITEM (channel, index, $e_1$, $e_2$, . . . . . ,$e_n$);

           value channel, index; integer channel, index;

           comment $e_1$, $e_2$, . . . ,$e_n$ may be:

               –    any integer, real or Boolean variable,
               –    any integer, real or Boolean array;


           begin procedure a (p); procedure p;

                begin p($e_1$); p($e_2$); . . . ; p($e_n$) end;
                GETLIST (channel, index, a)

           end

procedure PUTITEM (channel, index, $e_1$, $e_2$, . . . . ,$e_n$)

           value channel, index; integer channel, index;

           comment $e_1$, $e_2$, . . . . ,$e_n$ may be any of the above or any
           integer, real or Boolean expression;

           begin procedure a (p); procedure p;

                begin p($e_1$); p($e_2$); . . . . ; p($e_n$) end;
                PUTLIST (channel, index, a)

           end
```

3.2.2.3 INDEXED LIST INPUT AND OUTPUT ON STANDARD STORAGE MEDIA

GET (index, list)
PUT (index, list)

These procedures are the same as GETLIST and PUTLIST except that they always use channel 0, which specifies standard storage media. Information output to this channel is lost on termination of execution of the program.

3.2.2.4 WORD-ADDRESSABLE LIST INPUT AND OUTPUT

FETCHLIST (channel, address, list)
STORELIST (channel, address, list)

Channel is an integer parameter called by value and is the number associated with the file on the external device. Address is an integer variable called by name: on entry it contains the word address within the file where items are to be stored/retrieved; on exit it contains the word address of the next item in the file. List is a list procedure giving the items to be stored or retrieved.

These two procedures form a pair. The effect of STORELIST is to output to the output device the items presented by a call of the list procedure, in sequential words, starting from the item whose word address is given by address. The effect of FETCHLIST is to input from the input device values starting from the value whose word address is given by address and to assign them to the items presented by the list procedure.

The items presented by the list procedure can be of the following type:

1. an array – <u>real</u>, <u>integer</u> or <u>Boolean</u>

2. an arithmetic or Boolean expression.

On the external device, word addresses are sequential starting from 1. Each value (real, integer or Boolean) occupies 1 address. Values of different types may not be mixed on the same channel.

3.2.2.5 WORD-ADDRESSABLE ITEM INPUT AND OUTPUT

FETCHITEM (channel, address, $e_1, e_2, \ldots\ldots, e_n$)
STOREITEM (channel, address, $e_1, e_2, \ldots\ldots, e_n$)

These procedures can be defined in terms of FETCHLIST and STORELIST in a way entirely analogous to the definitions of GETITEM and PUTITEM in 3.2.2.2 above.

3.2.3 INPUT-OUTPUT PROCEDURES FOR BINARY SEQUENTIAL FILES

The procedures GETARRAY and PUTARRAY are provided for the retrieval and storage of arrays with binary sequential files.

GETARRAY (channel, destination)
PUTARRAY (channel, source)

Destination and source are the names of arrays.

GETARRAY 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. The record should contain the array arranged by rows (as defined in Section B.5, Array Transmission).

PUTARRAY 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. The record reflects exactly how the array is stored in memory, by rows.

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.
ERROR (key, destination)	is an execution-time trapping procedure. Key is an integer and is the key of the particular execution error to be trapped. Destination is a label. In the event of the keyed error occurring after a call of ERROR and being within its scope (see below), the destination is jumped to.

The scope of a call of ERROR is defined as the smallest block or procedure body surrounding the call plus any procedures called from this scope.

The values of key for ERROR are:

0	:	any error
1	:	overflow
2	:	array/switch error
3	:	parameter mismatch
4	:	standard function parameter errors
5	:	stack overflow

CHANERROR (channel, key, destination)	is an execution-time error trapping procedure for input-output entirely analogous to ERROR. Channel is an integer called by value.

The values of key for CHANERROR are:

0 :	any error
1 :	parity error
2 :	EOF
3 :	bad data
4 :	formatting errors
5 :	illegal operation
6 :	auxiliary procedure errors

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 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 forward past one end-of-file mark on coded or binary sequential file. It is treated as a dummy procedure on an output channel, on indexed and word addressable file.

SKIPB (channel)

This procedure spaces backwards past one end-of-file mark on coded or binary sequential file. 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 indexed or word addressable file, 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; 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 logical record on a sequential file, i.e., one unit record (print line or card) produced by coded sequential I/O or one record produced by a put array call. Backspace is only allowed on files of record type F, S or W or blocked files with one record/block.

3.5 MISCELLANEOUS PROCEDURES AND FUNCTIONS

IOLTH (channel)

This integer procedure returns as its value the length of the last item list read from or written to the external device associated with the value of the integer parameter channel. This procedure may only be used with indexed, binary sequential or word-addressable channels; use with other channels will give a zero result.

The length of the last item list is 1 if the item list was a single value and is the number of elements if it was an array.

POSITION (element, item)

This integer procedure returns as its value the displacement of the element (must be an array element) within the item (must be an array). This displacement, when added to the word address of the item, gives the word address of the element.

DUMP (identifying integer, option)

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 13). The dump is entitled

DYNAMIC CALL TO DUMP NUMBER < identifying integer > AT LINE < line number >

Identifying integer is an integer type variable

Option is an arithmetic expression taking on the same values as the post mortem dump options.

0	:	no dump
1	:	traceback of program execution
2	:	reduced octal dump of local quantities and formals
3	:	reduced symbolic dump (decimal) of local quantities and formals
4	:	complete symbolic dump (decimal) of local quantities, formals and arrays.

Note options 2-4 include the traceback.

The following convention has been added:

positive value of option :	dump of the dynamic chain of execution
negative value of option :	dump restricted to current block

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.

THRESHOLD (function, destination)

This is an environmental enquiry procedure. Function is an integer specifying the quantity to be accessed and destination is a variable of suitable type to hold the quantity. The available values for function are:

Function	Quantity to be accessed
1	Largest absolute integer
2	Largest absolute real
3	Smallest absolute real
4	Precision for unity

The largest integer N is defined as the representation in normalized floating point for which the following conditions are true:

$N+1 = N$

$N-1 \neq N$

This number is $2 \uparrow 48$ which is 281 474 976 710 656

The largest real number distinguishable from machine infinity is

$(2 \uparrow 48\text{-}1) * (2 \uparrow 1022)$ which is $1.2_{10}322$.

The smallest number distinguishable from zero by the machine in ALGOL 4.0 is $(2 \uparrow 47) * (2 \uparrow \text{-}1022)$ which is $\sim 1.6_{10}\text{-}294$. This is the smallest normalized number.

Precision for unity is defined to be the smallest real number that can be added to 1.0 in ALGOL 4.0 to produce a result distinguishable from 1.0. This number is $2 \uparrow \text{-}47$ which is $\sim 3.6_{10}\text{-}15$.

INRANGE (param)

This boolean function has one real value parameter that returns the value false if the parameter is infinite or indefinite, and true otherwise.

Specific representations exist for ± infinity and for ± indefinite. If such operands are used in arithmetic operations, a mode error will result. In ALGOL, infinite operands are caused by overflow and indefinite operands are the result of dividing zero by zero. Such operands also can be present in variables when storage space has been so preset and before the variable has been assigned a value by the program. Presetting of this kind may be either the result of the program loading operation or of object time stack space requests with the P-option on (Chapter 8).

MOVE (array 1, array 2)

This procedure moves the contents of array 1 to array 2. These arrays may be either normal or virtual arrays and they may reside either in CM [or SCM] or in ECS [or LCM] (Chapter 11). They must have the same dimensions. Any dimension errors will be trapped at execution time. When array 1 and array 2 are of different arithmetic types, the transfer is made without conversion.

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). A read operation immediately follows a write operation or vice-versa.

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 EFFICIENT USE OF FORMATTED INPUT-OUTPUT

The simplest procedures for formatted I/O are INPUT and OUTPUT. If the optimizing mode has been selected, an attempt is made to analyze and simplify calls to these two procedures at compile time and if an actual call satisfies all the conditions below, a faster call will be made at execution time:

1. The list of items to be input or output contains no procedure calls or formal parameters.

2. The format is valid and the format string is an actual string.

3. Each format item is a simple format item (see below).

4. The format does not contain the alignment mark J.

Using the definitions of format items in Chapter 3, Section A, the following are defined:

< simple number format > ::= < number format without insertion sequences >

< simple format item > ::= < simple number format > | < standard format > |

    < Boolean part > | < non format > | < S > | < A > | < insertion > | < alignment mark >

In general, a simple format item cannot contain associated or embedded alignment marks or insertion sequences or J or X.

Below are examples of simple format items:

4DD	+3ZD.2D -ZD	M
N	P	R
L	A	I
5S	'A STRING'	F
6B	↑	/
H		

The following are valid format items but are not simple format items:

```
'INTEGER' +ZD.2B 'FRACTION' 4D
              /F
              N/
            'ABC' 7S
```

The following declarations are assumed for the example given below:

real a, b, c; integer i, j;

Boolean p, q;

array X [1 : 6, 1 : 3], Y [0 : 5];

SIN and COS are standard functions.

Examples

Calls that satisfy conditions 1-4

(i) INPUT (60, ' ', a,Y[i]);

(ii) OUTPUT (2, '/ , 'ANSWER', 3B, +3ZD, ', ', F', a,p);

(iii) OUTPUT (7, '3(N),/', a, b, c *2+ 4.5);

(iv) OUTPUT (2, '(/, 3(+2ZD,3B))', Y);

Calls that do not satisfy all of the above conditions:

(v) INPUT (60, 'J,N', a);

(vi) OUTPUT (7, ' ', a, SIN (a), COS (a) *2.0);

(vii) OUTPUT (2, '/, 'ANSWER' 3B+3ZD,',',F',a,p);

Note that the results of examples (ii) and (viii) are the same; however, example (ii) will require less time to execute.

3.10 EXTENDED CORE STORAGE, LARGE CORE MEMORY PROCEDURES

The procedures, READECS and WRITEECS, are provided for reading from or writing to ECS or LCM. They are included only for compatibility with previous versions of ALGOL.

It is recommended that the procedure MOVE be used instead (Chapter 3, section 3.5).

READ ECS (ADDRESS, ARRAY, LABEL)

This procedure transfers the contents of successive locations from the user direct access area of ECS [or LCM] 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.

WRITE ECS (ADDRESS, ARRAY, LABEL)

This procedure transfers the contents of array, ARRAY, to the user direct access area of ECS [or LCM] 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.

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 to the compiler.

In the following definitions, the symbol eop indicates the delimiter 'EOP' which separates successive ALGOL programs.

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), plus the following definition of implicit outer block head.

4.1.1 SYNTAX

< implicit outer block head >::=< block head >;

< implicit outer block head list >::=< implicit outer block head >|< implicit outer block head list >
< implicit outer block head >

< pre >::=< any sequence of symbols except begin, code, algol, or
procedure >|< empty >

< post >::=< any sequence of symbols except eop>|< empty >

< source program >::=<pre > < program > < post > |< implicit outer block head> <program> < post>

< source program list >::=< source program >|< source program list > eop <source program >

4.1.2 SEMANTICS

A source program must contain declarations of all variables referenced in it. It must contain declarations for all procedures (except standard) it calls, including those that are compiled separately from the main program as ALGOL source procedures (Section 5.4.6., Chapter 2). However, the facility exists for adding implicit outer blocks to a program at compile time. This allows the user to reference identifiers in his program which are not explicitly declared therein but which will be present in the added outer blocks. A source library of such outer blocks can be maintained and included at compile time via control card options. Any number of such blocks may be added provided that the nested block level does not exceed the imposed limit.

Implicit outer blocks cause nesting of the source program. To ensure syntax correctness the compiler automatically supplies as many end's as necessary and issues a warning message telling that end's were missing.

Compilation of an ALGOL source program (generation of object code) starts with the 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 with the eop, whichever occurs first; however, a diagnostic is issued if the number of begin's is not equal to the number of end's.

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 (Chapter 2, Section 5.4.1).

4.2.1 SYNTAX

< pre >::=< empty >|< any sequence of symbols except begin, code, algol, or procedure >

< mid >::=< empty >|< any sequence of symbols except procedure, real, integer, or Boolean >

< 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 >

<external identifier >::=< identifier with less than or equal to 7 digits and/or letters >

< code identifier >::=< code number >|< external identifier >|< empty >

< code head >::=< pre > code < code identifier > ;< mid >|< pre > algol < code identifier > ; < mid >

< source procedure >::=< code head > ; < procedure declaration > ; < post >

< source procedure list >::=< source procedure >|< source procedure list > eop < source procedure >

4.2.2 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 procedure (Section 5.4.6., Chapter 2).

A source procedure may employ the same language features as a procedure declared in a source program.

Compilation of an ALGOL source procedure is initiated by the ALGOL symbol code ('CODE') or algol ('ALGOL'). These symbols are followed by the code identifier which is either a number in the range 0-99999, an external identifier of seven or fewer letters or numbers, or empty, and then by a semi-colon. In the case where it is empty the name declared in the procedure heading is taken to be the external identifier of the ALGOL source procedure. The same code number or external identifier is included in the body of the declararion 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 or algol symbol, compilation of the procedure starts normally, but an error message is issued and a code number of 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. If the eop occurs before the single statement is complete or before begin and end symbols balance, a diagnostic is issued.

The name generated for the procedure is the external identifier or CPXXXXX, where XXXXX is the code number, when the symbol code is used. 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 or external identifier results in 00000. When the code identifier is empty, the name of the procedure which is declared in the code body will be assumed to be the external identifier. 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.

A source procedure cannot contain overlays.

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.

The object program resulting from the compilation of a single source program, with no special binary subprogram input, is always executable, provided there are no compilation errors.

Source input for compilation must be on the standard system input file, described here only as cards.

Various SCOPE 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 APPENDIX C. 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

Source input file is constituted of cards in the sense of the standard system input file. Maximum card length is 126 characters.

An installation option can specify the number N of first characters of the card to be significantly interpreted by the compiler; any language structure may be across two or more cards provided it has no characters after N, since no syntactic meaning is attached to the characters appearing between N+1 and 126. At compile time, each card is counted and assigned a line count (beginning at 1) for reference by error messages. The line count is included in all source language listings as are characters beyond N.

The number of significant characters is established by default to 72, but it can be changed to another value by the K option of the ALGOL control-card (Chapter 6).

Example:

ALGOL,K=nn. $1 \leqslant nn \leqslant 126$

4.6 SOURCE DECK

A source deck consists of the cards which constitute one ALGOL source program or one ALGOL source procedure.

The source decks to be compiled are stacked consecutively, following the SCOPE control cards. The stack may contain any number of source programs or source procedures in any order within the restrictions described above. The source stack appears as one logical record on the input file.

If more than one source deck is submitted to the compiler, each source deck must be separated from the following by the delimiter, 'EOP'. This may be placed anywhere on a card; the following program must, however, begin on a new line. The last source deck must not be followed by an eop.

OUTPUT 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 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 compiler generates an object program which can be loaded for execution by the system loader. When the object program and its data requirement will not fit as a whole into available memory, a combination of the S option and the virtual and overlay delimiters should be used (see Chapters 6 and 10).

For each source procedure deck in the source input stack (Chapter 4), the compiler generates a SCOPE binary relocatable subprogram. A source program is generated as a SCOPE binary relocatable main program. These subprograms* are written out on the load-and-go file in accordance with SCOPE specifications.

After compilation, the load-and-go file 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 program 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, 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* contains an external name for any separately compiled source procedures or standard library or run-time system procedures called in that subprogram*.

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.

5.2 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 each card.

*or program

†See the reference manual for the COMPASS version available under the operating system.

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 1. Every 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 15 contains a description of system diagnostics.

ALGOL CONTROL CARD 6

The ALGOL compiler is called from the library by a SCOPE library card, which is the ALGOL control card.

The name ALGOL in columns 1–5 is optionally followed by a parameter list. If no parameter list is given, a period must follow ALGOL. If a parameter list is specified, it must conform to the control statement syntax for job control statements as defined in the SCOPE Reference Manual.

The parameter list is enclosed in parentheses or preceded by a comma and terminated by a period. The parameters are separated by commas and may appear in any order. All parameters must be fully contained on one card. Cards columns following the final terminator may be used for comments.

The general formats of the card are:

```
ALGOL (p1, p2, p3, . . . . . . . . . . . , pn)

ALGOL ,p1, p2, p3, . . . . . . . . . . . , pn.
```

Each installation may select default parameters which are assumed to be active when no conflicting parameters are given on the control card. In other words, the installation preset parameters need not appear on the control card. If a parameter other than the preset default parameter is desired, it must be explicitly selected;otherwise its absence is equivalent to its suppression. Except for Ø, where it has an intrinsic meaning,the writing of keyword = 0 means the explicit suppression of the corresponding parameter effects. This way must be used to remove an unwanted installation default parameter.

6.1 CONTROL CARD PARAMETERS

6.1.1 SYNTAX

< external identifier >::=< identifier with length less than or equal to 7 letters and/or digits >

< value >::=< digit >|< digit > < digit >|< external identifier >

< value list >::=< value >|< value list >/< value >

< keyword >::= I | U | L | A | R | B | P | Ø | C | E | N | D | F | S | K | X |

< parameter >::=< keyword >|< keyword > = < value list >|< empty >

< parameter list >::=< parameter >|< parameter list > , < parameter >

< control card >::= ALGOL, < parameter list > .| ALGOL (< parameter list >) | ALGOL.

Indeed this syntactic description does not show:

a) if a parameter need not be present it is not necessary to have an empty parameter between two commas.

b) the number of <value> in a <value list> is limited according to the keyword.

In any case this is definitely specified in the following section about semantics. A keyword may not appear more than once.

6.1.2 SEMANTICS

I Source input

I : source input is on standard input file (INPUT)

I = fn : source input is on file fn.

I = 0 : no source input.

L List control

L : list source program with fatal diagnostics on the standard output file (OUTPUT).

L = fn : list source program, fatal diagnostics on file fn.

L = 0 : list only fatal diagnostics on standard output file (OUTPUT)

R Cross reference map

R : produce and list a cross reference map at compile time, for the identifiers in the source program, on the file specified by the L option.

R = 0 : no cross reference map.

A Assembly listing

A : list the assembly language form of the object code on the file specified by the L option.

A = 0 : no assembly language listing.

N Advisory diagnostics.

N : listing of advisory diagnostics is performed in the file specified by the L option

N = 0 : list of advisory diagnostics is suppressed; only diagnostics fatal to code generation are listed.

Ø Optimization of generated code

Specifies the level of optimization the compiler is to perform.

Ø = 0 : compile program in the fast compile mode. This generated code is produced in the most general fashion without regard to special situations which can give rise to more efficient code.

Ø = 1 : performs linguistic optimization by optimizing procedure calling. Eliminates certain redundant compilations. For a further explanation of optimization, see chapter 12.

Ø = 2 : performs the optimizations of Ø = 1 and also subscript and <u>for</u> statement optimization.

U User implicit outer block head input supplementary to I, on standard file COMPILE.

U = fn : precede source program by the implicit outer block head list on file fn. Block heads precede the source program in their sequential order and must be in source form.

U = 0 : no file for implicit outer blocks.

For further information on implicit outer block heads, see Chapter 4.

C Comments interpretation for special delimiters. This option present requires the compiler to search comments for special delimiters interpretation.

These delimiters are:

a) debugging directives trace, snap, snapoff (Chapter 9)
b) overlay directive overlay (Chapter 10)
c) array bound checking directives checkon and checkoff

C = 0 : no comments interpretation.

C = 1 : debugging directives which are present in comments are detected by the compiler and cause debugging code to be inserted into the object program.

C = 2 : overlay directives which are present in comments are detected by the compiler and cause overlay directives in loader input format to be inserted into the object program.

C = 3 : array bound checking directives which are present in comments are detected by the compiler. The checkon directive causes array bound checking at execution time for each index of an array. All arrays are checked until the checkoff directive is encountered.

Multiple selection for the C option can be performed by separating each value by a slash. For example C = 3/2/1 is acceptable.

Note the virtual array directive is always detected by the compiler and is not dependent on the C option (see Chapter 11)

S Array storage allocation

S = 0 : all arrays are allocated to CM [or SCM].

S = 1 : virtual arrays are allocated to ECS [or LCM].

S = 2 : all arrays are allocated to LCM. Note this option applies only for programs to be executed on a CYBER 70/Model 76.

See Chapter 11 for an explanation of ECS/LCM usage

P Punch assembly language

P : punch assembly language form of the object code in standard assembly language card format on standard punch file (PUNCH).

P = fn : punch assembly language on file fn.

P = 0 : suppress assembly language punching.

P should not be used in overlay mode.

E Exit parameter

Abort the job to an EXIT control card if a fatal error occurs during compilation.

E = 0 : suppress abort in case of fatal error.

B Object program in standard relocatable binary load-and-go form.

B : output object program to standard load-and-go file (LGO)

B = fn : output object program to file fn.

B = 0 : no binary object program

D Dumpfile assignment

D : create the symbol file on standard dump file (DMPFILE).

D = fn : create the symbol file on file fn.

D = 0 : suppress the symbol file.

Note that to have a symbolic dump at execution time (option D $\geqslant$ 3 in Chapter 8), the symbolic file must be created at compile time.

F Fatal error termination

F : if a fatal error is found in the first pass (ALG1), terminate the compilation at the end of this pass.

F = 0 : continue until the normal end of compilation.

K Input record size

K = n : n is the number of significant characters to be interpreted by the compiler on the source card image.

The maximum number of characters that can be interpreted is K = 126.
The default value is K = 72.

X Real - Integer correspondence between formal and actual parameters (see Chapter 2, section 4.7.5.).

X = 0 : forbid any real-integer (or integer-real) correspondence between formal and actual parameters.

X or X = 1 : allow real-integer (or integer-real) correspondence between formal and actual parameters, and in the case real to integer perform the conversion.

Note that selection of this option will significantly degrade the performance of the program.

6.2 RESTRICTIONS AND ERRORS

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 outputs the following diagnostic to the dayfile:

```
CONTROL CARD ERROR : a b c d e f g h i j
```

where a b c . . . is the list of options having caused the error. A maximum of ten options is printed.

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 record of the channel card input file (see option C of Chapter 8); if C = 0 is selected on the card which calls for execution, no channel cards are read and only standard channels 60 and 61 (see 7.6.) will be defined; they are interpreted by the controlling routine before the object program is entered. The two types are: channel define and channel equate; 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 SCOPE file is defined by the channel card. The restrictions imposed by SCOPE 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, p_1, p_2, . . . , p_n

The eight characters CHANNEL, appear in columns 1–8 followed by a list of parameters; spaces are not allowed in the parameters and cause termination of the card.

Each parameter p_i, defined below, describes a different characteristic. Parameters are separated by commas. 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	SCOPE file name.

The parameter defining the type of file may be:

C	Coded sequential
B	Binary sequential
I	Indexed
W	Word-addressable

Only one of these may appear. The default option is C.

Note that if the file is specified as being binary sequential, no other parameters are available.

7.1.1 SERIAL FILE PARAMETERS

Any of the following parameters may be included when the C file type option or no file type option has been specified.

Pr	r indicates maximum line width; when omitted P136 is assumed.
PPs	s indicates maximum length of page (s lines). If PPO 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 $1 \leqslant b \leqslant r$

7.1.2 INDEXED FILE AND WORD–ADDRESSABLE FILE PARAMETERS

The following parameter may be included when the I or W file type option has been specified.

Ls	s indicates the size of the work storage area. The default value is 16 for indexed and 64 for word-addressable channels. A larger WSA will reduce the number of calls to SCOPE when each invocation of a standard I/O routine transfers a large amount of data.

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 previous channel define card. 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 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.

7.4 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.

7.5 STANDARD ALGOL CHANNEL CARDS

Two channel cards with standard channel numbers and characteristics are automatically supplied by the ALGOL system for the SCOPE standard input and output devices, as follows:

```
CHANNEL, 60 = INPUT, P80

CHANNEL, 61 = OUTPUT, P136, PP60
```

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.

7.6 TYPICAL CHANNEL CARDS

Some typical channel cards are:

```
CHANNEL, 35 = NUCLEAR, P120

CHANNEL, 47 = UNCLEAR, P400

CHANNEL, 29 = 35

CHANNEL,  4 = DISK, W, L200
```

EXECUTION-TIME OPTIONS 8

Execution-time options control the running of an ALGOL-60 object program. Options are specified by means of control cards.

Control card options are provided as parameters on the SCOPE control card (i.e., program call card) which initiates execution of the ALGOL program. It has the following format:

LGO (< options >) or EXECUTE (<options>) or lfn (<options>)

Each installation will have its own default values for these parameters.

The format of the options field follows:

< options > may be	< option-1 >{, . . . , <option k >}
option-i	is S {=0 | =1}
	or D {=0 |=n | =fn |=n/fn}
	or C {=0 |=fn}
	or P {=0 |=Z |=U}
	or T {=0 |=n }

8.1 STACK STATISTICS (S)

S option forms:

S = 0 : no stack statistics

S or S = 1 : output of stack statistics

This option allows the user to select either no output of statistics or the monitoring of the size of the stacks (SCM and LCM/ECS) during program execution. In the case of monitoring, upon termination (normal or abnormal) the amount of unused core in each stack is recorded on the standard output device. Monitoring will degrade the performance of the program.

8.2 ABNORMAL TERMINATION DUMP FORMAT (D)

D option forms:

D = 0 : no dump

D or D = 1 : traceback of program execution

D = 2 : reduced octal dump of local quantities and formals

D = 3 : reduced symbolic dump (decimal) of local quantities and formals

D = 4 : complete symbolic dump (decimal) of local quantities, formals and arrays

Note options 2 to 4 include the traceback.

The user may declare his own dump file by adding /fn. However fn must be the same file name as the dumpfile created during compilation. The default name is DMPFILE.

For example:

```
D = 4/MYDUMP
```

will use the dump file MYDUMP and in the case of abnormal termination will give a complete symbolic dump and traceback.

8.3 C, P, AND T OPTIONS

C Channel card input file.

C = 0 : no channel card input file

C : channel cards are read from file INPUT.

C = fn : channel cards are read from file fn.

P Array preset option.

P = 0 : no array presetting

P = Z : presets arrays to zeros

P = U : presets arrays to floating point undefined (177. . . 7)

The array space at each array declaration is preset; simple variables always are preset to 0.

Use of this option will degrade the performance of a program, especially if array activity is heavy.

T Debugging and dump output limit option

T = 0 : no debugging or dump output

T = n : causes debugging and dump output to be suppressed after the number of messages specified by n have appeared. Program execution will continue.

8.3.1 EXAMPLE

LGO (C,P=Z,D=3/MYDUMP)	Load LGO file. Read channel cards from INPUT file. Preset stack to zeroes. In case of abnormal termination, give traceback and a reduced symbolic dump using the file MYDUMP.

DEBUGGING FACILITIES 9

A source program may contain debugging directives which are designed to assist in the detection of faults during subsequent program execution. Under the normal mode of compilation these directives are ignored by the compiler and thus have no effect during execution of the resultant object code. However, if the debugging-option (Chapter 6, section 6.1.2, D-option) is selected any debugging directives which are present are selected by the compiler and cause debugging code to be inserted into the object program. The form of these debugging directives is such that they are acceptable ALGOL-60 character sequences. It is therefore unnecessary to remove them from a source program which one wishes to submit to an ALGOL compiler in which the directives are not recognizable.

9.1 GENERAL

The debugging directives are keyed to identifiers in the source program. There are two directives, trace and snap. The first of these, trace, monitors the flow of control through a program, whereas snap monitors changes in the values of program variables. Thus 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 snapoff 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. In this case the object program will include code which performs monitoring during subsequent execution by producing appropriate messages either on the standard output file or on any other channel selected, at execution time, to receive debugging messages (Chapter 8, section 8.2.). If an object program has been compiled under the debugging option then it is possible to suppress debugging output or to limit the amount thereof by selecting suitable options at execution time (Chapter 8, section 8.2.) but the source program should be recompiled in non-debugging mode in order to obtain an efficient object program or in order to select optimization options which are not permissible with the debugging mode.

The following sections describe in detail the syntax of debugging directives, their meaning and the format of the debugging messages which result from monitoring the various kinds of identifiers.

9.2 DEBUGGING DIRECTIVES

Debugging directives may be present in the source program at all stages of its development whether it is desired to activate them or not. 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 | snapoff
< debugging directive >::=< directive > <identifier list >
```

```
      e.g.          snap  a, b, marilyn
                    trace  L1
                    snapoff b
```

(N.B. trace, snap, snapoff, 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 the debugging directive after the last symbol in the sequence and immediately before the semicolon which terminates the sequence.

```
      e.g.          ; comment *** snap a, b;
                    begin comment first of all trace L1, Failure;
                    ; comment snapoff a ;
```

(N.B. directives will not be recognized in text following end)

If the debugging option is not requested at compile time, the previous example would be equivalent to semicolon, begin, semicolon. However, in debugging mode (Chapter 8, section 8.3, T-option) all comment structures are examined to detect the presence of trace, snap, or snapoff. The comment sequence does not have to contain a debugging directive, but if it does, a syntax error will result if the commentary sequence does not conform to the above definition of a debugging directive. Erroneously constructed debugging directives will not be recognized as such in the absence of the T-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.

9.3 TRACE DIRECTIVE

This directive monitors the flow of control within an object program. A trace directive is of the form

trace id_1 , ... , id_n

where the identifier list must contain at least one identifier. The order of the identifiers within the list is irrevelant 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 identifier list must each be declared in the same block as either 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 to, reference to, or call of a trace identifier while its associated declaration is within the current 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 (with a snapoff directive, 9.5).

The format of the messages produced by trace monitoring at execution time is described in the following sections referring to the various kinds of identifiers.

9.4 SNAP DIRECTIVE

This directive monitors changes in value of specified program variables during object program execution. A snap directive is of the form

snap id_1 , ... , id_n

where the identifier list contains at least one identifier. The order of the identifiers within the list is irrevelant and any member of the list will be referred to as a snap identifier. Unlike the trace directive, 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 must be declared in a block that embraces the block where the snap directive is to occur.

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, with its interpretation being made at compile-time only, the above definitions of the range of a snap directive are static (i.e., lexicographic) and refer to the physical program sequence rather than to the sequence in which the program is executed. This is more important in the next section in which a further method of terminating the effect of a snap directive is described.

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 for the various kinds of identifiers.

9.5 SNAPOFF DIRECTIVE

This directive provides an additional means of control over the monitoring process initiated by a snap directive. A snapoff directive is of the form

snapoff id_1, ... , id_n

where the identifier list contains at least one identifier. The order of the identifiers within the list is irrelevant. If any of the identifiers in the list is a snap identifier, the effect of the snapoff directive is to stop monitoring of that identifier in statements following the snapoff directive, unless the identifier appears in a subsequent snap directive. The snapoff directive is thus the logical opposite of the snap directive, and since it can appear in any part of the source program, it provides a statement-by-statement control of the monitoring of snap identifiers. It will be noted that, upon exit of the block in which a snap directive occurred, the effect of the directive terminates just as if a corresponding snapoff directive had appeared at the end of the block. The snapoff directive cannot be used to terminate monitoring of trace identifiers.

It should again be noted that the above definitions of monitoring ranges are physical and not logical, since the debugging directives are interpreted at compile time only.

Thus if a snapoff 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 thus be subject to monitoring in the statement subsequent to

a) the occurrence of the snap directive, or

b) the first declaration of the identifier,

whichever of a) or b) occurs the later; and in all lexicographically subsequent statements until

a) the occurrence of a snapoff directive for that identifier. or

b) exit from the block in which the identifier was declared

whichever of a) or b) occurs the earliest.

The following cases are worthy of note:

i) If a snap identifier occurs in a snap directive in a block in which the snap identifier is undeclared, then an advisory message is provided and the identifier is removed from the snap list.

ii) If it is desired 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, then it is necessary to insert the snap directive ahead of the procedure body (i.e., amongst the declarations in that block).

```
e.g.  begin

          integer i ;

          comment snap i ;

          procedure p(x) ; integer x ;

      begin i := i+x ; comment i is monitored ;

e.g.  if i = j then

          begin comment snap i ;

              i := complicated (x, y, z) ;

                  comment snapoff i ; j : = i + 1 end

          else . . .
```

9.6 DEBUGGING OUTPUT

The format and content of debugging output is dependent both upon the kind of variable being monitored and also upon the context in which it has been used. The information that may be obtained by applying debug directives is described in detail in the following sections categorized according to the kind of the monitored variable; however, a summary of the general formats of debugging output is given in this section.

A single item of debugging output will be referred to as a monitoring message. In the following list of monitoring messages the portions in lower case are replaced during execution by the appropriate 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 eight characters then only the first eight will appear in the monitoring message.

**LINE linenumber identifier AS PARAMETER
. . . produced for simple variables by snap

**LINE linenumber identifier [*] AS PARAMETER
. . . produced for arrays and subscript variables by snap

**LINE linenumber identifier : = value
. . . produced for simple variables by snap

**LINE linenumber identifier [*] : = value
. . . produced for subscripted variables by snap

**LINE linenumber GOTO identifier FROM LINE linenumber
. . . produced for labels by trace

**LINE linenumber CALL TO identifier FROM LINE linenumber
. . . produced for procedures by trace

**LINE linenumber EXIT FROM identifier
. . . produced for procedure by trace

In the following sections the appearance of a monitoring message of one of the above general formats is represented by a particular example.

9.7 LABEL MONITORING

When applied to a label identifier the trace directive produces monitoring messages during execution which indicate when and from where a transfer to that label occurred. If it is desired to monitor a label in this way, a trace directive referring to the label identifier must be inserted at some point in the block in which the label is declared, e.g., just before the label, or in the block head.

The flow of program control may pass through a label either by means of 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 means of go to statements will be monitored. This includes 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, as in the following example:

```
**LINE 231 GO TO INFEAS FROM LINE 120
```

In this case a transfer to the label INFEAS has occurred at line number 120 in the source program. INFEAS occurs at line 231.

9.8 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. In order to monitor a procedure in this way it is necessary to insert a trace directive at some point in the block in which the procedure is declared, e.g., prior to the procedure declaration in the block head.

Procedures may be invoked either by means of a procedure statement or by the appearance of a function designator (typed procedure) in an arithmetic or Boolean expression. Monitoring will be produced for the appropriate invokation of the trace identifier unless it is an external 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, as in the following example:

```
**LINE 250 CALL TO PRODUCT FROM LINE 512
```

In this case, the procedure PRODUCT, (declared in line 250) has been invoked from a statement on line number 512.

9.9 SIMPLE VARIABLE MONITORING

The snap directive permits monitoring of arrays and simple variables. In the case of simple variables, the snap directive is used to monitor changes, or possible changes, in the value of integer, real or Boolean variables. In order to monitor a variable, the associated identifier must be specified as a snap identifier according to the rules given in section 9.4. Monitoring can be terminated on a statement-by-statement basis by use of the snapoff directive, as described in section 9.5

The value of a simple variable may be changed if it is:

a) one of the left parts in an assignment statement, or

b) the controlled variable of a for clause, or

c) used as a call-by-name actual parameter corresponding to a formal parameter which undergoes a change of value.

The monitoring message for a direct assignment to a monitored variable as in cases a) and b) above 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 name, INDEX. In the above example the type of the monitored variable is integer. If the variable type is real then the value is output in decimal standard format. If the variable type is Boolean then the output representation is 'TRUE' or 'FALSE'.

Monitoring of changes in value caused by actual/formal correspondence as in case c) above 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, then its value will change whenever the corresponding formal variable is altered, as in cases a), b), c), above. 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 message or messages produced depends upon the context of the usage of the snap identifier as a parameter since it cannot always be guaranteed that control will be returned to the code that will monitor the final value of the variable. Thus the following example of usage of the monitored variable as an actual parameter in a procedure statement will produce the following message:

```
comment check the determinant, snap DET;
INVERT (array, DET);
comment DET is the determinant of array, snapoff DET;
```

The following type of message will always be produced:

```
**LINE 102 DET AS PARAMETER
```

Example:

```
comment snap M, N;
     M := 2 * unstack (N) + 1;
```

would produce the following messages

```
**LINE 115 M := 11
**LINE 115 N AS PARAMETER
```

9.10 ARRAY AND SUBSCRIPTED VARIABLE MONITORING

The snap directive, when applied to array identifiers, is mainly used to monitor occasions on which the elements of the array may have been altered. Since the element or elements concerned are generally dependent on the execution-time value of one or more subscript expressions, the monitoring usually supplies only the point of usage.

One exception is the case of direct assignment to the array element as a result of its appearance as a left part in an assignment statement. Thus the following source text:

```
comment snap A;
A [ j, 3 * j - 2 ]  := 2 ** 5;
```

would produce the following message:

```
** LINE 127 A [*]  := 32
```

It will be noted that the monitoring message contains the new value of the subscripted variable whether it be 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 kind subscripted variable, respectively. As in the case of simple variable parameters, the use of an array or subscripted variable is considered to be capable of changing the array values if the actual parameter consists of either 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 A[*] AS PARAMETER
**LINE 53 C[*] AS PARAMETER
```

It will be noted that no attempt is made to record the values of an array in the monitoring messages since the elements altered are in the general case not easily established at compilation time. The monitoring process is therefore primarily concerned with simply identifying critical usages of array identifiers rather than the detailed results of those usages. The only exception to this rule is that of the direct assignment to an element, in which case the new value is provided.

It is suggested that if preliminary use of the snap directive on an array identifier is not sufficiently detailed or too indiscriminate, then a more selective monitoring should be attempted by use of the snap directive on a temporary variable introduced to sample critical array elements at the points indicated by the original monitoring of the array itself.

OVERLAYS 10

The overlay declaration provides the capability of overlaying blocks so that a program may be executed in less memory than would otherwise be needed. From a code generation point of view, the only effect of an overlay declaration is to alter the memory allocation scheme of the translated statements.

Prior to execution, the sections of an overlay program are linked by the loader and placed on a mass storage device or tape file in their absolute form, thus no time is required for linking at execution time. However, one of the effects of using overlays in a program is that execution time is increased. This is because each time an overlay is to be executed, it must first be loaded into memory.

10.1 OVERLAY DECLARATIONS

10.1.1 SYNTAX

< primary level >::=< digit >|< digit > < digit >

< secondary level >::=< digit >|< digit > < digit >

< overlay parameter part >::= (< external identifier > , <primary level> , < secondary level >)

< overlay declaration >::= <u>overlay</u> <overlay parameter part >

< declarative comment >::= <u>comment</u> <overlay declaration>;

10.2 EXAMPLE

<u>comment</u> <u>overlay</u> (ovfile,1,0);

10.3 SEMANTICS

Primary level may be any positive integer number in the range 1:63 inclusive. Secondary level may be any positive integer number in the range 0:63 inclusive. The main overlay is defined by the system and contains the outermost block of the program as well as the run-time system. Therefore, the user must not declare the main overlay.

In any case, a maximum of three overlay levels, including the main overlay is allowed. The external identifier is the file name onto which the loader is creating the absolute program. The same file name must be used in all overlay declarations. Overlay declaration applies only to blocks and procedure bodies (which always constitute a block), thus the comment containing the overlay declaration must appear after the <u>begin</u> of the block to be overlayed.

The overlay declaration in a program must conform to the following rules:

1. A secondary overlay must be a block nested in a primary overlay block.

2. An overlay cannot be a block nested in an overlay block of the same level.

A primary overlay is defined by setting

a) <primary level> $\neq 0$ and

b) <secondary level> = 0.

A secondary overlay is defined by setting

a) its <primary level> equal to the primary level of the primary overlay it is attached to and

b) its <secondary level> $\neq 0$

For example:

<u>overlay</u> (myfile,1,0) defines a primary overlay and
<u>overlay</u> (myfile,1,1) defines a secondary overlay of the preceding primary

All primary overlays are stored at the same point immediately following the main overlay (0,0). Therefore, the loading of a primary overlay will destroy the preceding primary overlay.

A secondary overlay is stored immediately following its primary overlay. Again, the loading of a secondary overlay will destroy the preceding secondary overlay.

Note that the SCOPE control cards are different for programs with overlays and for those without overlays. The user is referred to the SCOPE Reference Manual.

10.4 RESTRICTIONS

A code procedure cannot contain overlay declarations.

10.5 EXAMPLES

Example 1 :

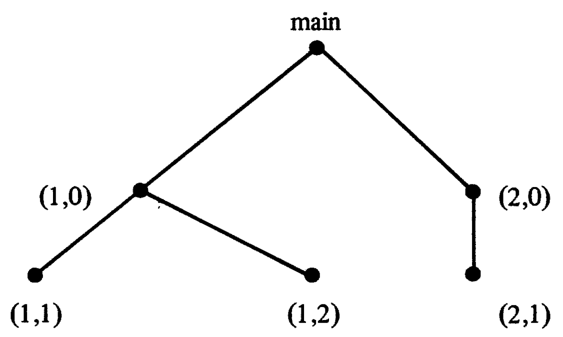
This example demonstrates a simple, direct way of programming overlay structures and produces the most efficient execution time overlay handling.

Di	is a declaration
Si	is a statement

```
begin comment block 1;
     D1; S1;
  begin comment block 2;
       comment overlay (F1, 1, 0);
       D2; S2;
       begin comment block 3;
            D3;
            comment overlay (F1,1,1);
            if expr1 then goto L;
            S3
       end of overlay (1,1);
       begin comment block 4;
            comment overlay (F1,1,2);
            D4;S4
       end of overlay (1,2)
  end of overlay (1,0);
L: begin comment block 5;
       comment overlay (F1,2,0);
       D5; S5;
       if expr2 then
```

```
      begin comment block 6;

            comment overlay (F1,2,1):

            D6;S6

      end of overlay (2,1)

   end of overlay (2,0)

end
```

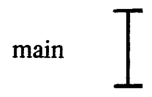
The preceding program structure consists of two primary level overlays; the first contains two secondary level overlays; the second refers only to one secondary level overlay.



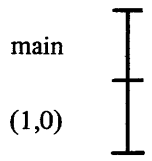
The order of execution of the blocks and the structure of memory at execution time are as follows:

Execution Phases | **Memory Map for Generated Code**

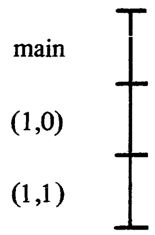
Execution of main is started.



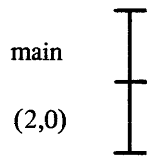
Block 2 is initiated and overlay (1,0) is loaded.



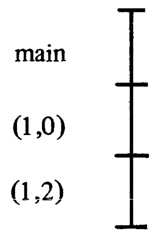
Block 3 is initiated and overlay (1,1) is loaded.



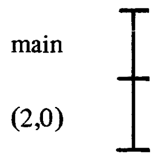
If exprl is true, then block 3 execution is ended and block 5 is initiated causing overlay (2,0) to be loaded.



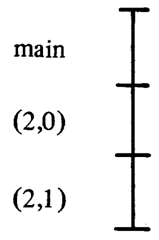
Otherwise block 3 is executed to its end, and block 4 is started, causing overlay (1,2) to be loaded.



At the end of block 4, block 5 is started, causing overlay (2,0) to be loaded.



If expr2 is true, then block 6 is initiated which results in the loading of overlay (2,1), otherwise the program is ended.

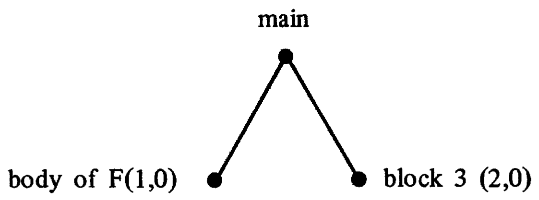


Example 2 :

This example demonstrates a more sophisticated and time consuming use of overlays.

```
begin integer J;

     procedure F(N); integer N;

     comment procedure body in block 2;

     comment overlay (File,1,0);

     begin J:=N/2;

          if J > 0 then F(J-3);

     end of procedure body and overlay (1,0);

     begin comment block 3;

          comment overlay (File,2,0);

          integer K;

          K:=5;

          F(K**2)

     end of overlay (2,0)

end
```

The program consists of two primary level overlays.



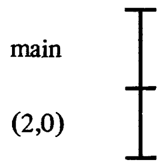
Procedure F is declared in main. The procedure body, however, is a primary level overlay; it will be loaded only when procedure F is executed.

The order of execution of the blocks and the structure of memory at execution time are as follows:

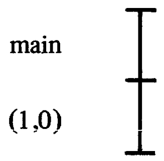
Execution Phases

Memory Map for Generated Code

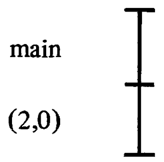
Execution starts with main, then overlay (2,0) is loaded. A call is made to procedure F.



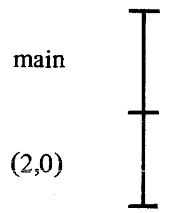
Procedure body is loaded overlaying block 3.



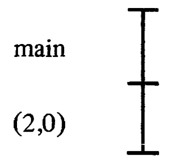
When executing J:=N/2 in procedure F, block 3 is loaded, overlaying the procedure body to evaluate the called-by-name formal parameter N whose actual parameter originates from block 3.



Overlay (1,0) is then reloaded to continue procedure execution.
Further recursive executions of F will only refer to elements local to main and the procedure body, and will therefore cause the loading of no other overlays.



When F(K**2) is evaluated, return to block 3 is executed and overlay (2,0) is loaded one more time.



USE OF LCM AND ECS 11

11.1 GENERAL

By a combination of the S option and the virtual comment delimiter, the user is given control of the allocation of his arrays between SCM and ECS [or LCM]. An array can be declared to be a virtual array, the primary function being to enable the more effective use of ECS. A virtual array can be referenced only as a whole, not by individual elements and is permitted only as a parameter to a procedure, in particular to the MOVE procedure (See section 3.5, Chapter 3). For example, if A is a virtual array and B is a normal array, the procedure MOVE (A,B) will move the entire array A into B, where it can be modified and then returned by MOVE (B,A). Please note that the S = 1 option will allocate all virtual arrays to ECS [or LCM]. If S = 0 is selected, all virtual arrays will be stored in SCM, but still will remain accessible only as a whole.

The S option controls the allocation of arrays (See Chapter 6).

S = 0 : all arrays are allocated to CM [or SCM].

S = 1 : virtual arrays are allocated to ECS [or LCM].

S = 2 : all arrays are allocated to LCM. Note this option applies only for programs to be executed on a CYBER 76.

11.2 VIRTUAL ARRAYS

When a <u>virtual</u> comment directive is encountered, all arrays in the immediately following array declaration will be virtual. When a virtual array is used as a formal parameter, the virtual comment directive must precede the array specification in the procedure heading.

A <u>virtual</u> array is incompatible with an <u>own</u> array.

A <u>virtual</u> array used as a formal parameter cannot be specified to be a <u>value</u> array.

11.3 EXAMPLE

In the following example A is declared to be a virtual array and B is declared to be a normal array. The sole use of a virtual array is as a parameter to a procedure, either a normal procedure (shown by P(A)) or the special procedure MOVE. In this example, the normal array B is moved to the virtual array A.

The allocation of A and B to SCM or ECS/LCM, depends upon the previous selection of the S option.

```
begin comment virtual;array A [1:5,1:5];
    array B [1:5,1:5];
    procedure P(A); comment virtual;array A;
        begin;
        MOVE (B,A)
        end;
    P(A)
end
```

OPTIMIZATIONS 12

The compiler optionally performs four kinds of optimization while the runtime system is itself organized to perform optimally at all times. In addition, the compiler is provided with a set of standard vector functions which allow the user to optimize source programs at the algorithm level and benefit from certain hardware.

12.1 LANGUAGE DEPENDENT OPTIMIZATIONS

Machine independent optimizing involves subscript variable handling, the calling of procedures and the handling of formal variables and special treatment of the procedures INPUT and OUTPUT.

12.1.1 SUBSCRIPTS

For subscript variables the source program is delimited into nodes, for statements and blocks, in each of which code is to be generated so that as many subscript computations as possible will share common code while still retaining the explicit statement ordering. The largest gains from this obviously come from the for statement in which as many hidden computations as possible are removed out of the loop. To permit this however, it is necessary to re-compute the values of those removed sequences when any of the dependent operands is changed. Such changes can occur at assignment statements and in procedure calls due to assignment to actual parameters or by global side-effect. For dealing with procedure calls a side-effect table is constructed before code generation starts and this is consulted after every procedure call in a for statement.

12.1.2 PROCEDURES

Procedure calling can be time consuming at runtime in ALGOL because of the necessity to check the formal specifications against the actual parameters. The optimizing of this aspect involves an examination of all the calls to any procedure to determine parameter correctness. Simultaneously the set of all actual parameters corresponding to each formal is found which permits special case code to be generated in the procedure body for all call-by-name formals. The ALGOL Report states that the procedure bodies are to be considered as being inserted in the program at calls with the actual parameters replacing the formals as in macros, and of course this would lead to optimal code at each call. This compiler does not do that; it provides one copy of the code of each declared procedure but generated so that the actual parameters of the calls are all treated in the least general manner which accommodates them all. This is not true of pre-compiled procedures in which each call-by-name formal is given the most general treatment.

12.1.3 SIMPLE FORMATTED I/O

The formatted I/O procedures INPUT and OUTPUT are the most commonly used I/O routines and the compiler examines their calls to determine if the parameters obey certain restrictions. If they do then calls are generated to special library routines which execute faster than the general case. The restrictions which will produce these calls are first that the format string include no items with insertion sequences (see Chapter 3, section A.1.3.2) and that the data parameters be simple local variables or arrays but not subscripted variables or formals.

12.2 VECTOR FUNCTIONS

These procedures available in standard library are designed to take advantage of the hardware instruction stack. Without the stack, on model 72, the gain from using the vector functions over programmed loops performing the same operations is about a factor of 5. With the stack, on models 74 and 76 the gain is of the order 10 or more. In all cases, the vectors concerned need to be of size 10 or greater before the gains are appreciable, owing to initializations performed within the functions. The vector functions are described in Appendix D.

12.3 EFFICIENT PROGRAMMING FOR RUN TIME PERFORMANCE

ARRAYS: within each block like arrays of the same size should be declared in a single declaration.

PROCEDURES: Actual parameters should agree in type with the formal specifications.(The ALGOL Report states they should anyway but this compiler allows real actual parameters to correspond to integer formals.) Integer actuals and real formals are also allowed but performance is not affected in this case.

PROCEDURES INPUT AND OUTPUT: The format parameter should be a literal string in which numeric items do not contain insertion sequences. The data parameters should be local, simple variables or arrays,and not either formals or expressions.

GENERATED MACHINE CODE: It is not possible to influence this directly but the code is obviously a function of the preceding optimizations.

THE VECTOR FUNCTIONS: When the program is handling vectors and arrays these functions should be employed wherever possible in place of programmed loops. Overall gain in total program execution time of a factor of 6 is possible with suitably structured algorithms in amenable programs on models 74 and 76.

LCM/ECS USAGE: Arrays situated in SCM on the model 76 are faster to access than those in LCM. For optimal usage of LCM use should be made of the virtual array concept for the arrays situated in LCM while declaring just sufficient normal arrays in SCM, for working purposes. This requires careful programming to keep track of the contents of work arrays at any time. On models 72 and 74 only virtual arrays can be in ECS and only considerations of size dictate its use (see Chapter 11).

OBJECT-TIME ABNORMAL TERMINATION DUMP 13

Upon detection of a fatal error during program execution, the ALGOL control routines perform the following actions:

(1) Empty all output format areas onto their associated files.

(2) Print the appropriate diagnostic on the standard output file – i.e., channel 61.

(3) Print a structured dump.

See Chapter 15 for a list and description of the object-time diagnostics.

The amount of information given by (3) is determined by the execution-time option parameter D (see Chapter 8).

13.1 STRUCTURED DUMP

The structure dump traces back the execution path from the point where the error occurred through the block structure to the entry point of the program. The information relevant to the program will be selected in accordance with the D option.

The following information will be available:

(1) The line number at which the error occurred.

(2) The line number and name in which each active block was declared. This name may be the procedure name, the code procedure identifier, the external procedure identifier or the standard procedure name if the block is a standard procedure.

Code procedures are always dumped in octal.

13.2 ENVIRONMENTAL INFORMATION

Environmental information consists of values of formal (by value) and/or local variables belonging to the block currently being dumped. Formal variables appear only if the particular block is a procedure.

Simple local variables and simple formal parameters are represented by their values.

Arrays may also be dumped according to the D option.

For boolean variables the values *T* (true), and *F* (false) will be printed.

Note that values not yet affected will contain garbage.

13.3 CROSS–REFERENCE LISTING

The user may optionally request a cross-reference listing of the identifiers used in his program. This listing is a useful debugging aid in conjunction with the error messages. The identifiers are listed in the order in which they are declared.

For each non-standard identifier, the following information is supplied:

Name of the identifier

Type : real, own, integer . . . . .

Kind : array, procedure, switch . . . . .

For arrays : number of dimensions

For procedures : number of formal parameters

Line number of identifier declaration or specification

Identifier block number

List of line numbers which reference the identifier.

For each standard identifier, the list of line numbers that reference the identifier is supplied at the beginning of the cross-reference list.

OBJECT-TIME DESCRIPTION 14

14.1 OBJECT PROGRAM AND STACKS

14.1.1 RUN-TIME SUPERVISORY PROGRAM

A Run-Time Supervisory program, ALGORUN, controls the object-time execution of an ALGOL program. In this function it acts as the interface between the compiled program and the SCOPE operating system. Certain functions are shared between the compiled program and ALGORUN (e.g., stack handling), whereas others (e.g., overlay handling) are solely the province of the latter.

Calls to the Run-Time System and Library are treated as references to externals during compile time and are satisfied by the loader at load time.

14.1.2 OBJECT CODE STRUCTURE

The object-time form of each main program or code procedure consists of a program block and several labeled common blocks. The program block contains the block map and the code for the program itself. The labeled common blocks contain constants, strings, labels, and procedure constants, the <u>own</u> variables, and a special parameter area. The object code is generated forward, from the first <u>begin</u> to the last <u>end</u>, and forms the SCOPE binary relocatable file.

In overlay mode, the program block is organized according to the overlay structure specified, each overlay being preceded by the overlay directive to the loader. The labeled common blocks are attached to the main overlay.

14.1.3 LIBRARY SUBPROGRAMS

The Library subprograms obey the same structural rules as a binary subprogram generated by the compiler. Each standard procedure is represented in the library by one subprogram. In overlay mode, a required subprogram is attached to the earliest overlay that requests it.

14.1.4 SCALAR SPACE

The "scalar space" of a block includes storage for all those items which can be quantified at compile time, such as simple variables declared in the block, array descriptors, dope vectors and various compiler generated locations. It does not include storage for subscripted variables.

14.1.5 PROGRAM DEPTH AND PROGRAM LEVEL

The "program depth" of a block is the number of nested procedure declarations within which the block is contained.

A "program level" is the largest set of blocks of the same depth, such that one block of the set encloses all of the other blocks (if any) of the set. The enclosing block is called the "first block" of the set and has the property that its depth is one greater than the depth of its own enclosing block (if any). The program level is the unit of allocation of scalar space for blocks, such that the relative positions of scalar space for all blocks within a program level are determined at compile time. The address to which the variables in a program level are relative is called the "relocation base".

"Program level zero" is the unique program level whose constituent blocks have a zero program depth. The "first block" of program level zero is the outermost block of the main program.

14.1.6 OBJECT TIME STACKS

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 overlap variables which are not simultaneously active. The overlap process is described below.

During execution of an object program, all variables are contained in variable-length memory stacks consisting of 60-bit entries, one or more pertaining to each variable. Since the stacks include only active entries, their sizes fluctuate.

The SCM stack is located in SCM and contains the scalar space for program levels, the actual parameters and return information for procedures and the SCM-resident arrays.

The LCM/ECS stack (if it exists) is located in LCM or ECS. It only contains the virtual arrays (see Chapter 11). Non virtual arrays may be also allocated in LCM under the S storage option (see Chapter 6).

14.1.7 REFERENCING THE STACKS

Every reference to simple variables in the SCM stack is generated as the relative position of the variable in its program level plus an index register which contains the appropriate relocation base.

When a call to a procedure is made, the environment of the procedure has to be established because this call constitutes an entry into a new program level. The entry into the new program level is achieved by assigning the first available (in-active) position in the stack to the relocation base for the new program level and by requesting the necessary amount of stack for the scalar space of the complete program level (i.e., including all its blocks). Also a "display" is built into the stack, containing the relocation bases for all the statically enclosing program levels.

When a new block is entered, its position within the program level in which it is contained is known and its scalar space requirements have already been reserved. It only remains to physically record its stack limit before handing control to the block's code.

Array storage in SCM is physically assigned starting at the first available position in the SCM stack after the allocation for a complete program level, and its stack limit is recorded in the block where the arrays are declared. Arrays here are indirectly addressed through array descriptors and dope vectors contained in the scalar space of their declaring block.

Arrays in LCM/ECS are also indirectly addressed through array descriptors and dope vectors in the scalar storage of their declaring block (i.e., located in SCM stack). Array storage in LCM/ECS is assigned starting at the first available position in the LCM/ECS stack and its stack limit is also recorded in the declaring block in SCM stack.

When a block is exited, the space in the stacks (SCM and LCM/ECS) occupied by its local arrays is released, so that it may be allocated by a new stack space request, however the amount of SCM stack occupied by its scalar space can only be utilized by a parallel block in the same program level.

When a procedure is exited, the scalar space of the complete program level is then 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.

14.1.8 OWN VARIABLES

All own quantities are assigned entries in the labeled common block area. Own variables are treated as global in definition (local to the whole program), though they are only local to the block in which they are declared, just like other variables.

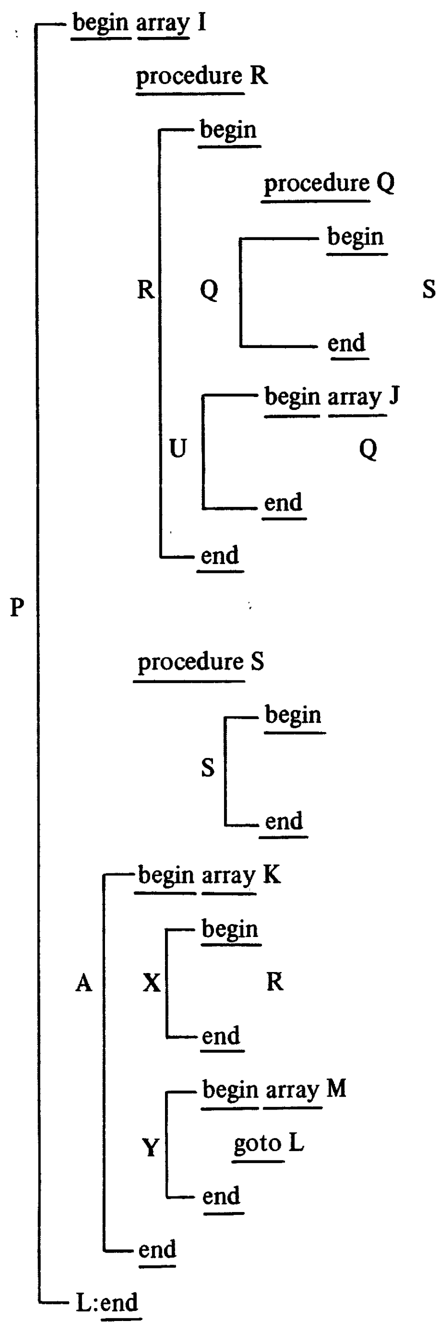
14.1.9 STACK LISTING

The compiler assigns a block number to each block in the program and constructs the BLOCK MAP which contains information about the block and procedure structure of the program for use during traceback.

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.

14.1.10 EXAMPLE

Consider the following program outline:



```
  ┌─ begin array I
  │      procedure R
  │         ┌─ begin
  │         │       procedure Q
  │         │         ┌──── begin
  │        R│   Q     │                  S
  │         │         └──── end
  │         │      ┌──── begin array J
  │         │     U│            Q
  │         │      └──── end
  │         └─ end
 P│
  │      procedure S
  │            ┌─ begin
  │           S│
  │            └─ end
  │     ┌─begin array K
  │     │    ┌─ begin
  │    A│   X│        R
  │     │    └─ end
  │     │    ┌─ begin array M
  │     │   Y│     goto L
  │     │    └─ end
  │     └─ end
  └─ L:end
```

Block P is the program itself; block A and procedures R and S are at the same level within P; block U and procedure Q are contained in procedure R at the same level. Blocks X and Y are at the same level within block A.

Block X contains a call to procedure R; block U a call to procedure Q, and procedure Q a call to procedure S; block Y contains a jump to label L within the outermost block. Blocks P, U, A and Y declare respectively arrays I, J, K, and M. The changes in the stack and in the displays can be visualized in figure 14.1.

Following the entry into the program, the stack is already reserved for blocks A, X and Y, although it is not yet used. The array I follows in the stack. The display at P indicates that only program level P is accessible by having its relocation base address p.

On entry to block A, array K is allocated after I in the stack. When procedure R is called, space for all of its program level is reserved after K and the display in R indicates that both p and r are accessible. The compiler knows which blocks are accessible (as opposed to program levels, which are dynamic) and since space for them is always prereserved, it becomes unnecessary to include block information in the displays.

On entry to U, array J is placed after U in the stack. When procedure Q is called, its display contains p, r and q. However when procedure S is entered, its display only contains p, and although R, U, J and Q are still in the stack, they become inaccessible.

On exiting S and releasing its stack space, R, U, J and Q become active automatically, since the display at Q already exists. When eventually R is exited, the display at P will indicate that only p is accessible.

On entering block Y, the compiler knows that space is already available and only allocates stack for array M. 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.

14.2 STACK ENTRIES

14.2.1 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 format (Section 5.1.3, Chapter 2)

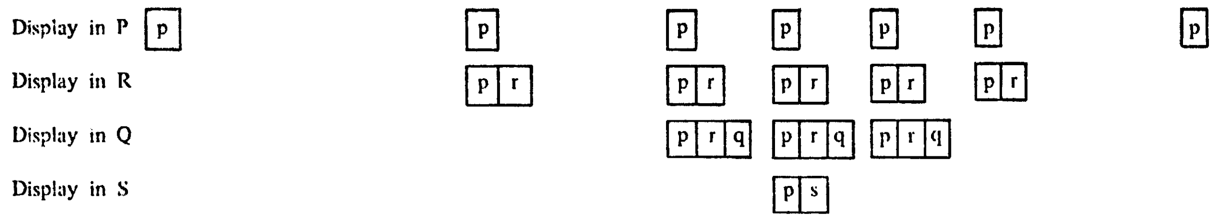
INTEGER

60-bit entry in standard floating point format (Section 5.1.3, Chapter 2)

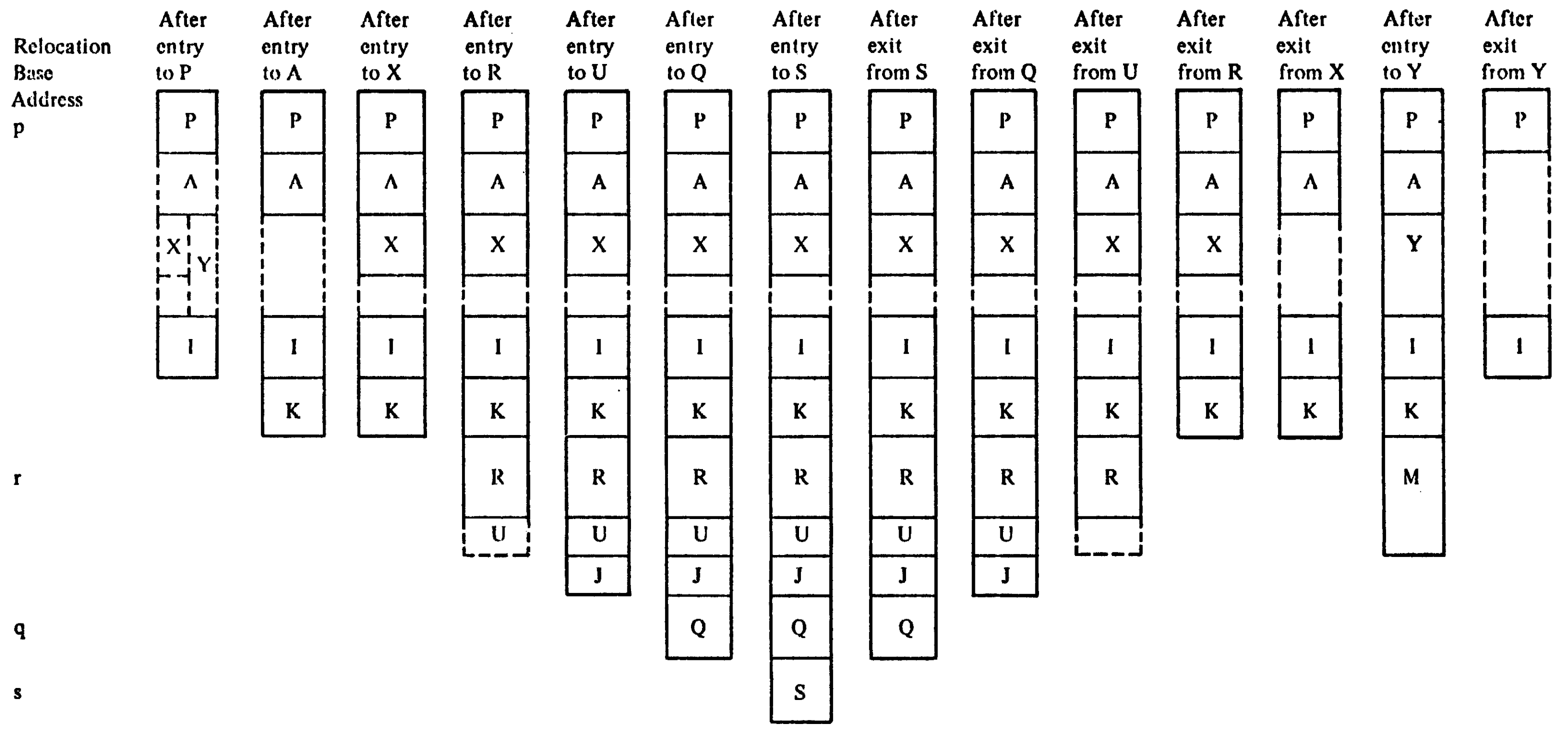
BOOLEAN

60-bit entry in which bits 58-0 are irrelevant and bit 59 is set to 1 for true and 0 for false.

DISPLAYS AND THEIR CONTENTS AT THE SAME STAGES



STACK CONTENTS AT DIFFERENT STAGES DURING PROGRAM EXECUTION



14.2.2 DESCRIPTION OF VARIABLES

All descriptions of variables in the stack have the following general form:

xxxx	xxxxxx	xxxx	xxxxxx
$x=0: <x>_1 <i>_1 <o>_3 <k>_4 <t>_3$	address 1	$<s>_1 <o>_4$ $<\text{overl}>_7$	address 2
$x=1: <x>_1 <\bar{i}>_1 <7>_3 <\bar{k}>_4 <\bar{t}>_3$			

x is the transform bit; if $x = 1$ no transformation is required and the following 11 bits are complemented
if $x = 0$ a transformation from <u>real</u> to <u>integer</u> is necessary

i is the expression bit; if $i = 1$ no expression evaluation is required
if $i = 0$ expression evaluation is necessary

s is the storage bit for arrays, virtual arrays and subscripted variables

k is the kind as follows:

k	Kind	Possible use
00	Switch	
01	String	
02	Label of designational expression	
03	No-type procedure	Formal and local
04	Type procedure	
05	Array	
12	Virtual array	
06	Constant	
07	Expression	
10	Simple variable	
11	Subscripted variable	Formal only
13	No-parameters, no-type procedure	
14	No-parameters, type procedure	

t is the type of the variable

t	Type	Possible use
0	No type	Formal and local
1	Integer	
2	Real	
3	Boolean	

overl is an index to the overlay table (when necessary)

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.

14.3 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.

14.3.1 TERMINOLOGY

All references to the stack in the object program are relative to the beginning of the stack area for a particular program level (i.e., procedure). When a program level is entered at execution time, the base address of the corresponding stack area is assigned. This absolute base address is

the relocation base RRRRRR of the program level (and of all of the blocks it encloses).

the term program address PPPPPP, means an address pointing to a position in the object program.

the term stack address, AAAAAA, means an absolute address pointing to a particular stack entry.

the term common address, CCCCCC, means an absolute address pointing to a particular labelled common area.

14.3.2 DESCRIPTION

Return Information for Expressions

xxxxx	xxx	xxxxxx	xxxxxx
LINE	OVERL	RRRRRR	PPPPPP

PPPPPP	Program address of next instruction following the evaluation of the expression.
RRRRRR	Relocation base of the program level where the expression evaluation is requested.

OVERL	Index of the overlay where the expression evaluation is requested.
LINE	The source line number where expression evaluation is requested.

Return Information for Procedures

xxxx	xxxxxx	xxxx	xxxxxx
Number of formals	RRRRRR	OVERL	PPPPPP

RRRRRR	Relocation base of the program level where the call is made
PPPPPP	Program address of next instruction following the procedure call
OVERL	Index of the overlay where the call is made

00 Switch

xxxx	xxxxxx	xxxx	xxxxxx
5777	NNNNNN	0000	AAAAAA

NNNNNN	Number of elements in the switch list
AAAAAA	Stack address for the first element in the list

The descriptions of the labels or designational expressions (see kind 02 below) which constitute the switch list is as follows:

$<$ Designational expression of the nth switch element $>_{60}$

$<$ Designational expression of the (n-1)th switch element $>_{60}$

aaaaaa $<$ Designational expression of the 1st switch element$>_{60}$

01 String

xxxx	xxxxxx	xxxx	xxxxxx
5767	$<C>_1<T>_1<X>_6<N>_{10}$	0000	CCCCCC

$<C>$ format flag ; if $<c>=1$, the string has been analyzed and can be used as a format string

if $<c>=0$, the string must be analyzed

$<T>$ translation flag ; if $<T>=1$, the string has been translated

$<X>$ X Replicator count

$<N>$ Number of characters in the string

CCCCCC Address of first word of the string in a labelled common

02 Label

xxxx	xxxxxx	xxxx	xxxxxx
5757	AAAAAA	OVERL	PPPPPP

AAAAAA	Block header's address (in the stack) of the block where the label is declared
OVERL	Index of the overlay where the label is contained
PPPPPP	Program address of the instruction corresponding to the label

02 Designational Expression

xxxx	xxxxxx	xxxx	xxxxxx
5757	AAAAAA	OVERL	PPPPPP

AAAAAA	Block header's address (in the stack) of the block where the designational expression is contained
OVERL	Index of the overlay which contains the expression
PPPPPP	Program address of the code which evaluates the expression and jumps to the resulting address

03 No-type procedure

xxxx	xxxxxx	xxxx	xxxxxx
5747	RRRRRR	OVERL	PPPPPP

and

04 Type procedure

xxxx	xxxxxx	xxxx	xxxxxx
573$\bar{t}$ 204t	RRRRRR	OVERL	PPPPPP

RRRRRR	Relocation base of the program level where the procedure is declared
OVERL	Index of the overlay where the procedure is declared
PPPPPP	Program address of the code for the procedure

05 Array

xxxx	xxxxxx	xxxx		xxxxxx
572$\bar{t}$ 205t	AAAAAA	S	0000	DDDDDD

AAAAAA	Stack address where the quantity FWA-LBE is contained
FWA	is the base address of the array elements in stack
LBE	is the lower bound effect for the array
DDDDDD	Stack address of the dope vector (see below)
S	Flag indicating if the array is in LCM (=1) or in SCM (=0) (Bit 29)

The elements of an array are assigned after the last location reserved for the program level where the array's declaring block is contained. <u>Own</u> arrays are handled in the same way, except that their elements are assigned among the <u>own</u> variables.

The elements of an array called by value are copied (and transformed as necessary) to a position after the last location reserved for the program level from where the array is called.

The dope vector for the array declaration

array A $[L_1:U_1, L_2:U_2, \ldots, L_n:U_n]$ is:

$$
\left.
\begin{array}{lll}
 & < & C_n >_{60} \\
 & < & C_{n-1} * C_n >_{60} \\
 & & \cdots \\
 & < & C_2 * C_3 * \ldots\ldots * C_{n-1} * C_n >_{60} \\
 & < & \text{LENGTH} = C_1 * C_2 * \ldots\ldots * C_{n-1} * C_n >_{60} \\
\text{DDDDDD} \rightarrow & < & \text{Lower Bound Effect} = \text{LBE} \quad >_{60} \\
 & < & \text{Number of dimensions} = n \quad >_{60} \\
\begin{array}{l}\text{This part of}\\ \text{dope vector}\\ \text{exists only}\\ \text{when array}\\ \text{bound check-}\\ \text{ing is required}\end{array}
\left\{
\begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array}
\right. &
\begin{array}{l} < \\ < \\ < \\ < \\ \\ < \\ < \end{array} &
\begin{array}{r} L_1 >_{60} \\ U_1 >_{60} \\ L_2 >_{60} \\ U_2 >_{60} \\ \cdots \\ L_n >_{60} \\ U_n >_{60} \end{array}
\end{array}
\right\}
\begin{array}{l}\text{Dope Vector}\\ \text{Size} = 3n + 2 \text{ words}\end{array}
$$

$$
\begin{array}{lll}
 & & \cdots \\
 & & \cdots \\
\text{AAAAAA} \rightarrow & < & \text{FWA} - \text{LBE} >_{60}
\end{array}
$$

where $C_i = U_i - L_i + 1$

and

$$\text{LBE} = (((\ldots(L_1 * C_2 + L_2) * C_3 + L_3) * \ldots) * C_n + L_n$$

The address of any element A $[i_1, i_2, \ldots, i_n]$ is calculated as follows:

$$\text{address} = \text{FWA} + (((\ldots i_1 * C_2 + i_2) * C_3 + i_3) * \ldots) * C_n + i_n - \text{LBE}$$

For example, the declaration array A [1:3, 2:5] has a dope vector:

```
           <                       4>60
           <                      12>60
dddddd→    <                       6>60
           <                       2>60
           <                       1>60
           <                       3>60
           <                       2>60
           <                       5>60
```

and a descriptor

```
<5725     aaaaaa     0000     dddddd>60
```

```
where aaaaaa→   <                 FWA - 6>60
```

06 Constant

xxxx	xxxxxx	xxxx	xxxxxx
{571t̄ / 206t}	000000	0000	CCCCCC

CCCCCC	Common address at which the constant is found.

07 Expression

xxxx	xxxxxx	xxxx	xxxxxx
{770t̄ / 007t}	RRRRRR	OVERL	PPPPPP

RRRRRR	Relocation base of the program level where the expression is situated.
OVERL	Index of the overlay containing the expression
PPPPPP	Program address of the code that evaluates the expression

10 Simple Variable

xxxx	xxxxxx	xxxx	xxxxxx
$\left\{\begin{matrix}567\bar{t}\\210t\end{matrix}\right\}$	000000	0000	AAAAAA

AAAAAA	Stack address of the variable

11 Subscripted Variable

xxxx	xxxxxx	xxxx		xxxxxx
$\left\{\begin{matrix}766\bar{t}\\011t\end{matrix}\right\}$	RRRRRR	S	OVERL	PPPPPP

RRRRRR	Relocation base of the program level where the code to evaluate the address is found
OVERL	Index of the overlay containing the code
PPPPPP	Program address of the code which computes the address
S	Flag indicating if the variable is in LCM (=1) or SCM (=0) (Bit 29)

12 Virtual Array

xxxx	xxxxxx	xxxx	xxxxxx
$\left\{\begin{matrix}565\bar{t}\\212t\end{matrix}\right\}$	AAAAAA	0000	DDDDDD

The only difference between this kind of array and the SCM array described above is that the elements of the array are assigned in LCM or ECS

AAAAAA and DDDDDD	are as described above

13 No-parameters No-type procedure

xxxx	xxxxxx	xxxx	xxxxxx
$\left\{\begin{matrix}7647\\0130\end{matrix}\right\}$	RRRRRR	OVERL	PPPPPP

RRRRRR	Relocation base of the program level where the expression's code to call the procedure is found.
OVERL	Index of the overlay containing the expression's code
PPPPPP	Program address of the code where the procedure call is made.

14 No-parameters Type procedure

xxxx	xxxxxx	xxxx	xxxxxx
$\left\{\begin{matrix} 76\bar{3}\bar{t} \\ 014t \end{matrix}\right\}$	RRRRRR	OVERL	PPPPPP

RRRRRR, OVERL and PPPPPP are as described for kind 13.

ALGOL DIAGNOSTICS 15

Two types of diagnostics are issued by the ALGOL compiler system: compiler generated diagnostics and those generated at object time.

15.1 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 number, which is printed as part of the source listing, and each diagnostic inlcudes the line number of the source card in error and a summary of the error condition.

The diagnostics are grouped under the following headings:

LEXICOGRAPHIC AND SYNTACTIC DIAGNOSTICS

PRE-SEMANTIC DIAGNOSTICS

SEMANTIC DIAGNOSTICS

PROCEDURE CHECKING WARNINGS (OPTIMIZING MODE)

GOTO AND LABEL DIAGNOSTICS

SYMBOLIC FILE CONSTITUTION WARNINGS – FILE SUPPRESSED

A heading appears only when a corresponding diagnostic occurs.

Compiler diagnostics are either alarms or advisory messages; alarms cause the generation of object code to be suppressed but advisory messages do not.

15.1.1 COMPILER ALARMS

A compiler alarm indicates that a serious error has been found in the source text and causes the suppression of code generation regardless of any user request, although normal compilation and error checking continue until the end of the source text. Source errors will cause otherwise legal text to become invalid but the compiler will attempt to localize the effect of each individual error.

The error messages are self-explanatory. Upon detection of an error, the diagnostic is dynamically constructed with additional relevant information inserted into the message (such as identifier name, identifier type, operand name, syntax structure, line number, etc.). In the following examples the underlined parts indicate the additional information inserted into the diagnostic:

LINE 123 : ILLEGAL (AFTER , IN ARRAY DECLARATION

LINE 456 : BB SPECIFIED PROCEDURE IS ILLEGAL BY VALUE

LINE 789 : UNRECOGNIZED COMPOUND DELIMITER ≠SKIP≠ . . . IGNORED EXCEPT LAST ≠

For a lexicographic error, the illegal character is replaced by a blank and the resulting operands are processed according to the current state of the compiler. This often results in a second diagnostic being issued.

Example:

The statement ≠INTEGER≠ H$I;

produces the following diagnostics:

LINE 22 : ILLEGAL CHARACTER $ OCCURS IN COLUMN 34 . . . ACTS AS A LEXICOGRAPHIC SEPARATOR

LINE 22 : MISSING DELIMITER BETWEEN H and I . . . FIRST OPERAND DELETED

There are two error messages related to the creation of the dumpfile:

ERROR IN BLOCK TABLE and

NO SYMBOL FILE FOR CODE PROCEDURE

These diagnostics indicate an internal failure in the compiler and should be brought to the attention of CONTROL DATA.

15.1.2 RECOVERY MESSAGES

In some cases the error message is preceded by RECOVERY : to indicate a fatal error has been detected and the compiler has attempted to correct the error and continue the compilation. However, it is still considered a fatal error. Furthermore, it is possible that the compiler has guessed incorrectly, in which case the recovery might have provoked additional diagnostics which do not necessarily indicate the presence of other errors. The following are examples of recovery diagnostics:

LINE 15 : RECOVERY : ILLEGAL = AFTER ≠FOR≠ REPLACED BY :=

LINE 120 : RECOVERY : ≠QRRAY≠ REPLACED BY ≠ARRAY≠

15.1.3 ADVISORY MESSAGES

Advisory messages do not necessarily indicate the presence of an error in the source text but provide information which may be useful in detecting errors not recognized as language infringements.

For example:

```
LINE  55  :  ADVISORY  :  NON-FORMAT STRING

LINE  92  :  ADVISORY  :  DELIMITER(S) BEFORE PROGRAM START
                          May indicate a missing ≠BEGIN≠

LINE 115  :  ADVISORY  :  ERROR IN DUMP FILE WRITING
                          Indicates a hardware error; DMPFILE is suppressed and compilation continues.
```

The advisory messages, particularly NON-FORMAT STRING, can be a nuisance. They can be suppressed, however, using the option N=0 on the control card. Fatal errors will always be output.

15.1.4 ADVISORY MESSAGES IN OPTIMIZING MODE

When optimization is requested, as indicated in Chapter 6, additional checking is performed upon the use of procedures and parameters. In the case of a type or kind mismatch between an actual parameter and a formal parameter, an advisory message is issued.

```
LINE 104  :  ADVISORY  :  KIND ERROR IN ACTUAL PARAMETER

LINE 189  :  ADVISORY  :  ACTUAL PARAMETER CANNOT CORRESPOND TO LEFT-HAND SIDE
                          USAGE OF FORMAL.

LINE 237  :  ADVISORY  :  ACTUAL PARAMETER SHOULD BE A PROCEDURE WITH PARAMETERS
```

In non-optimizing mode, as well as for the cases listed above, the correspondence between actual and formal parameters will be checked at run-time. Any parameter mismatch will only lead to an error if the procedure call is executed.

15.1.5. COMPILER STOP

For a violation of an implementation restriction, a compiler stop occurs. A diagnostic indicating the nature of the error is printed on the output file, followed by a message indicating the compiler stop.

15.1.6 COMPILE ABORT

If the compiler should fail, the following message will be printed on the dayfile:

```
    COMPILER ABORT IN LINE ——— SCAN ——— .
```

Indicates an internal error in the ALGOL compiler and should be brought to the attention of CONTROL DATA.

15.2. OBJECT-TIME DIAGNOSTICS

Upon normal exit from an object program, the contents of all non-empty format areas are output.

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. If asked for at execution time, information which traces the execution path through the currently active block structure and stack information for each block are then printed on the standard output device (see D option, Chapter 8).

15.2.1 DUMP SYMBOL FILE DIAGNOSTICS

In case of an error involving the dump symbol file, the following messages are issued:

● on the dayfile

- UNKNOWN ACTION IN SYMBOL FILE	File given as symbol file in D option has not been created or it has been partially destroyed.
- I/O ERROR DURING DUMP	A hardware error has occurred during dump on output file or DMPFILE.
- STOP DUMP TOO LONG	Each call to dump is limited to approximately 6 pages of printing.

● inside the dump printout

***** SYMBOL FILE NOT PROVIDED- OCTAL DUMP REPLACING	Option D must be explicitly set at compile-time to get a symbolic dump at run-time.
***** WRONG SYMBOL FILE - OCTAL DUMP REPLACING	The symbol file has not been created at the last compilation of the program or pertains to another program.
***** ERROR IN SYMBOL FILE -	Hardware error in DMPFILE during reading.

15.2.2 ERROR CONTROL PROCEDURES

ERROR and CHANERROR are execution-time error trapping procedures which permit the user to regain control after the detection of an error of the same type as the given key (Chapter 3, section 3.3). In the following list of diagnostics a key is given for each message. A call to ERROR or CHANERROR will trap all errors with the corresponding key.

15.2.3 OBJECT-TIME DIAGNOSTICS

Object-Time Diagnostics		ERROR KEY	CHANERROR KEY
ALPHA FORMAT ERROR	Output value is too large.		4
ALGOL I/O ERROR	I/O error detected by SCOPE on ALGOL channel.		5
ARITHMETIC OVERFLOW	Evaluation of an expression results in an arithmetic error (e.g., if the operand is infinite). This is the same as a mode 2 error as defined by SCOPE.	1	
ARRAY DIMENSION ERROR - DIMENSION NO.:	Number of dimensions in actual parameter array in procedure call differs from number in formal parameter array.	3	
ARRAY LOWER BOUND ERROR - DIMENSION NO.: VALUE:	Computed element address in an array is not within lower array bound.	3	
ARRAY UPPER BOUND ERROR - DIMENSION NO.: VALUE:	Computed element address in an array is not within upper array bound.	3	
BAD DIAGNOSTIC LIB: ERNUM:	This indicates an internal error in ALGOL. This information should be sent to CONTROL DATA CORPORATION.		
EXPONENTIAL PARAMETER ERROR	Argument of EXP procedure is too large.	1	
FETCH ITEM/LIST TYPE ERROR	Attempt to mix variable types on word addressable channel in FETCH ITEM/LIST call.		3
FORMAT ITEM ERROR	More characters in expanded format item than permitted in INPUT, OUTPUT, INLIST, and OUTLIST.		4
FORMAT MISMATCH	Format item and corresponding I/O item have incompatible types or kinds.		4
FORMAT REPLICATOR ERROR	Replicator in call to FORMAT procedure not in proper range.		4

Object-Time Diagnostics		ERROR KEY	CHANERROR KEY
FORMAT STRING ERROR	Incorrect format string.		4
GET ITEM/LIST TYPE ERROR	Type of data does not match variable on GET ITEM/LIST call.		3
H/V LIM ERROR	H LIM AND V LIM arguments L, R, and L', R' out of range.		6
ILLEGAL CHANNEL NUMBER VALUE :	A negative, undefined or infinite channel number was found in procedure CHANERROR.		5
ILLEGAL KEY VALUE:	The key used in procedure ERROR or CHANERROR is not defined. It is changed to zero and execution proceeds.		5
ILLEGAL OPERATION - SEQUENTIAL FILE / INDEXED FILE / WORD-ADDRESSABLE FILE / BINARY SEQUENTIAL FILE	An attempt has been made to use an incompatible I/O operation on the type of channel specified in the diagnostic (e.g., calling GETLIST on a sequential file).		5
ILLEGAL STRING INPUT	Attempt to read into a string parameter during a call to INPUT or INLIST.		4
ILLEGAL STRING OUTPUT	Illegal character in string output.		4
INPUT KIND ERROR	Input destination is neither simple variable, array, or subscripted variable.		3
INSUFFICIENT SCM FOR OVERLAY - OVERLAY INDEX :	The indicated overlay requires more SCM.	5	
LAYOUT CALL ERROR	Procedures established by H END and V END and label set by NODATA are not accessible after return from the layout procedure called by INLIST or OUTLIST.		6
LCM/ECS ARRAY SIZE ERROR	Computed LCM/ECS array size is negative or zero.	2	

Object-Time Diagnostics		ERROR KEY	CHANERROR KEY
LCM/ECS STACK OVERFLOW	Required array space in LCM/ECS exceeds available memory.	5	
LOGARITHM PARAMETER ERROR	Argument to LN procedure may not be negative or zero.	4	
NEGATIVE SWITCH INDEX	Value of switch designator is negative.	2	
NON-FORMAT INPUT	In non-formats I, R, L, or M, input field contains non-octal characters.		3
NUMBER SYNTAX	Number input in standard format does not conform to proper syntax.		3
NUMERIC INPUT ERROR	Data input under format control does not conform to numeric input format.		3
OPERATION ON ACTIVE CHANNEL	An attempt has been made to input or output to an indexed channel while that channel is active.		5
OUTCHARACTER ERROR	Parameter to OUTCHARACTER call is not in proper range.		4
OVERLAY FILE NOT PRESENT OVERLAY INDEX :	The file for the indicated overlay is not present.	0	
OVERLAY NOT FOUND ON FILE OVERLAY INDEX :	The indicated overlay cannot be found on the declared file.	0	
PARAMETER COUNT ERROR	Number of actual parameters in procedure call is incorrect.	3	
PARAMETER KIND ERROR - PARAMETER NO. :	Kind of actual parameter in procedure call does not correspond to kind of associated formal parameter.	3	
PARAMETER NOT STRING NOR INTEGER ARRAY	A formal I/O string can only be matched by an actual string or array.	3	
PARAMETER TYPE ERROR PARAMETER NO.	Types of actual and formal parameters in procedure call do not correspond.	3	

Object-Time Diagnostics		ERROR KEY	CHANERROR KEY
SCM ARRAY SIZE ERROR	Computed SCM array size is negative or zero.	2	
SCM STACK OVERFLOW	Data requirements of program exceed available memory.	5	
SIN - COS ERROR	Argument to SIN or COS procedure is too large.	1	
SQUARE ROOT PARAMETER ERROR	Argument to the SQRT procedure may not be negative.	4	
STORAGE INCOMPATIBILITY FOR ARRAY	Storage allocation (SCM or LCM/ ECS) in an actual array differs from that of the format array in a procedure call.	3	
STRING ELEMENT ERROR	Rules of STRING ELEMENT violated.	6	
SWITCH BOUNDS ERROR	Value of switch designator out of range.	2	
SYSPARAMETER ERROR - F -	SYSPARAM called with incorrect F parameter.		6
SYSPARAMETER ERROR - Q -	SYSPARAM called with incorrect Q parameter.		6
TABULATION ERROR	Argument of TABULATION not in proper range.		6
THRESHOLD ERROR	Request for invalid function to standard procedure THRESHOLD.	4	
UNASSIGNED CHANNEL	No channel defined for channel number used in program.		5
UNCHECKED EOF	End-of-file mark detected, but no provision made with EOF procedure.		2
UNCHECKED PARITY	No PARITY procedure for parity error detected.		1
ZERO SWITCH INDEX	Value of switch designator is zero.	2	

STANDARD SCOPE 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	072	6	6	41	6	06	6	066
A	A	01	12-1	61	12-1	101	7	7	42	7	07	7	067
B	B	02	12-2	62	12-2	102	8	8	43	8	10	8	070
C	C	03	12-3	63	12-3	103	9	9	44	9	11	9	071
D	D	04	12-4	64	12-4	104	+	+	45	12	60	12-8-6	053
E	E	05	12-5	65	12-5	105	–	–	46	11	40	11	055
F	F	06	12-6	66	12-6	106	*	*	47	11-8-4	54	11-8-4	052
G	G	07	12-7	67	12-7	107	/	/	50	0-1	21	0-1	057
H	H	10	12-8	70	12-8	110	(	(	51	0-8-4	34	12-8-5	050
I	I	11	12-9	71	12-9	111	)	)	52	12-8-4	74	11-8-5	051
J	J	12	11-1	41	11-1	112	$	$	53	11-8-3	53	11-8-3	044
K	K	13	11-2	42	11-2	113	=	=	54	8-3	13	8-6	075
L	L	14	11-3	43	11-3	114	blank	blank	55	no punch	20	no punch	040
M	M	15	11-4	44	11-4	115	, (comma)	, (comma)	56	0-8-3	33	0-8-3	054
N	N	16	11-5	45	11-5	116	. (period)	. (period)	57	12-8-3	73	12-8-3	056
O	O	17	11-6	46	11-6	117	≡	#	60	0-8-6	36	8-3	043
P	P	20	11-7	47	11-7	120	[	[	61	8-7	17	12-8-2	133
Q	Q	21	11-8	50	11-8	121	]	]	62	0-8-2	32	11-8-2	135
R	R	22	11-9	51	11-9	122	%††	%	63	8-6	16	0-8-4	045
S	S	23	0-2	22	0-2	123	≠††††	" (quote)	64	8-4	14	8-7	042
T	T	24	0-3	23	0-3	124	→	_ (underline)	65	0-8-5	35	0-8-5	137
U	U	25	0-4	24	0-4	125	∨	!	66	11-0 or	52	12-8-7 or	041
V	V	26	0-5	25	0-5	126				11-8-2†††		11-0†††	
W	W	27	0-6	26	0-6	127	∧	&	67	0-8-7	37	12	046
X	X	30	0-7	27	0-7	130	↑	' (apostrophe)	70	11-8-5	55	8-5	047
Y	Y	31	0-8	30	0-8	131	↓	?	71	11-8-6	56	0-8-7	077
Z	Z	32	0-9	31	0-9	132	<	<	72	12-0 or	72	12-8-4 or	074
0	0	33	0	12	0	060				12-8-2†††		12-0†††	
1	1	34	1	01	1	061	>	>	73	11-8-7	57	0-8-6	076
2	2	35	2	02	2	062	≤	@	74	8-5	15	8-4	100
3	3	36	3	03	3	063	≥	\	75	12-8-5	75	0-8-2	134
4	4	37	4	04	4	064	¬	^ (circumflex)	76	12-8-6	76	11-8-7	136
5	5	40	5	05	5	065	; (semicolon)	; (semicolon)	77	12-8-7	77	11-8-6	073

†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.

††††The ≠ character replaces the apostrophe in the new character set.

A

INTERFACE MACROS B

A number of COMPASS coded macros are provided to expand the areas of application of ALGOL-60 programs. ALGOL programs can make use of subprograms coded in COMPASS (by using code) to perform tasks which require the compactness of extreme code efficiency.

These macros are subdivided into four groups: Entry/Exit Macros, which provide the link between the ALGOL declaration of a procedure and the COMPASS subprogram; Specification Macros, which supply the means for analyzing the parameters passed to the COMPASS subprogram; Formal Handling Macros, which enable certain operations with these parameters; and a macro for requesting local stack space.

No restrictions are imposed upon register use, since all registers used by the Run Time System are restored whenever they are needed within the macros. Care must be taken, however, to save any registers that may be destroyed as a result of a macro call.

The following text uses the symbols { , , } to signify that there is a choice between the items enclosed within the braces and separated by commas.

B.1. ENTRY/EXIT MACROS

The following two macros 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 calling program.

B.1.1 ALGOL

This macro provides a linkage between the code declaration of an external procedure in an ALGOL main program and the corresponding COMPASS subprogram. Control will transfer to the ALGOL 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 program level in the stack and establishes the environment for the macros.

```
name   ALGOL   number
```

name	Optional location field parameter, up to 7 characters. A 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	Optional, up to 5 digits. The code number used in the declaration of the associated procedure in the ALGOL main program. This number, given as CP ↱ number, links the COMPASS subprogram with the ALGOL declaration.

If more than 7 characters are used for the name or more than 5 digits for the number, they will be truncated and a warning message given. The name and the number are related as follows:

name	**number**	**traceback name**	**link for loader**
blank	blank	CP00000	CP00000
blank	number	CP↦ number	CP↦ number
name	blank	name	name
name	number	name	CP↦ number

Examples:

ALGOL declaration	Entry Macro		
procedure A(x,y); code 0;		ALGOL	
procedure A(x,y); code;	A	ALGOL	
procedure A(x,y); code 100;		ALGOL	100
	AAA	ALGOL	100
procedure A(x,y); code PROCA;	PROCA	ALGOL	

B.1.2. RETURN

This macro causes a normal return from a COMPASS coded procedure. Although the procedure may terminate in other ways (GOTO, ERROR, and XEQ macros, for instance) this will be the normal point of exit, and will return control to the point in the main program from which the ALGOL 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. Additionally, it performs an optional conversion of the contents of the X register.

```
RETURN   xreg,type
```

xreg Optional X register designator. The X register contains the value of the function represented by the preceding code in accordance with the type. If xreg is omitted, it is assumed the code was executed from a procedure statement.

type Optional { R,I,B,F } . If none given, the default is R. The contents of xreg are assumed to be:

- (R) Floating point and will be normalized
- (I) Floating point and will be an integer† (in the ALGOL sense, i.e., entier (xreg+0.5))
- (B) boolean and will be passed straight through without any additional operation
- (F) Machine integer† and will be converted to floating point and normalized

†Throughout this appendix a distinction is made between the term machine integer as used in COMPASS, and the ALGOL type integer, which is stored internally in floating point.

Examples:

RETURN	If called as procedure
RETURN X4	If called as function. The result in X4 is assumed to be floating point and will be normalized.
N EQU 5 RETURN X.N,B	If called as function. Result in X5 is boolean (bit 59 determines value).

B.2. SPECIFICATION MACROS

The following macros are used within a COMPASS coded procedure to establish the number, kinds, and types of actual parameters in the current invocation of the procedure.

B.2.1. PARAMS

This macro specifies the number of actual parameters in the current invocation of the COMPASS procedure. No check is performed on the parameter count.

```
PARAMS    xreg
```

xreg	X register designator. The X register will receive the number (in machine integer form) of actual parameters transmitted in the current call.

Example:

```
PARAMS    X5
```

B.2.2. KIND

This macro provides a numerical representation of the kind of an actual parameter. No check is made on the parameter.

```
KIND    param,xreg
```

param	{Absolute address expression, X register designator}. The value or contents of param is the ordinal of the actual parameter to be examined; 2 indicates the second parameter, for example.

xreg X register designator. The X register will receive the numerical kind code (in machine integer form) of the specified actual parameter. The register may be the same as param. The codes corresponding to each kind of actual parameter are:

switch	0	simple variable	8
string	1	subscripted variable	9
label	2	virtual array	10
no type procedure	3	no-parameter, no-type procedure	11
typed procedure	4	no-parameter, typed procedure	12
array	5		
constant	6		
expression	7		

Examples:

```
FORMAT  EQU     5
        SX2     4
        KIND    FORMAT,X6
        KIND    3,X1
        KIND    X2,X3
        KIND    X2,X2
```

B.2.3 TYPE

This macro provides a numerical representation of the type of an actual parameter. No check is made on the parameter.

```
TYPE    param,xreg
```

param { Absolute address expression, X register designator } . The value or contents of param is the ordinal of the actual parameter to be examined.

xreg X register designator. The X register will receive the numerical type code (in machine integer form) of the specified actual parameter and may be the same register as param.

no type	0
<u>integer</u>	1
<u>real</u>	2
<u>boolean</u>	3

Examples:

```
TYPE    2,X5
TYPE    X3,X3
TYPE    ACTUAL,X7
```

B.2.4. SPEC

This macro permits the programmer to restrict the kinds of actual parameters. Since allowable kinds for a given parameter may be run-time dependent, this check occurs in-line. This macro may produce abnormal termination with a diagnostic, and it does not provide direct information about the kind of the actual parameter. Any number of parameters may be checked with a single call to this macro, the only limit is imposed by COMPASS, which allows up to 9 continuation lines per statement.

```
SPEC    (PAR1:SPC1, PAR2:SPC2, . . . , PARn:SPCn)
```

PAR1	{Absolute address expression,X register designator}.
PAR2 . . . PARn	Absolute address expression. The value or contents of PARi is the ordinal of the actual parameter to be examined. Only PAR1 may be an X register, the second and subsequent PARi must be absolute address expressions.
: (colon)	Separates PARi part from the SPCi part.
SPCi	String of up to 12 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
string	G
label	L
no type procedure	P
typed procedure	F
array	A
constant	C
expression	X
simple variable	V
subscripted variable	S
virtual array	R
no-parameter, no type procedure	N
no-parameter, typed procedure	T

Since there are 13 kinds, the string may have up to 12, to avoid checking when all kinds are allowed.

If the actual parameter is not of the specified kinds, the diagnostic PARAMETER KIND ERROR is produced and the program will abort (See 3.3).

Examples:

SX1 SPEC	2 (X1:CXVST,5:VS)	The second actual parameter must be a constant, expression, simple variable, subscripted variable or typed procedure without parameter; the fifth parameter must be a simple variable or subscripted variable.
SPEC	(3:AV,X2:W)	The macro call is erroneous because the second pair contains an X register.

B.3. 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 the same parameter may be checked several times if it is used by different macros.

B.3.1. VALUE

This macro will cause the current value of a parameter to be computed. It is applicable to the following kinds of parameters only: constant (6), expression (7), simple variable (8), subscripted variable (9) or no-parameters-typed procedure (12) (i.e., CXVST). Optionally, the parameter may be checked for kind.

VALUE param,xreg,kind,check

param	{ Absolute address expression, X register designator } . Indicates the ordinal of the parameter.
xreg	X register designator. The X register will contain the value of the indicated actual parameter. The value may be a floating point quantity in the case of <u>real</u> or <u>integer</u>. If <u>boolean</u> only bit 59 is used (see 5.1.3).
kind	Optional, absolute address expression. Numerical kind code of the actual parameter, if known. This specification will produce a more efficient expansion. It will be ignored if its value cannot be ascertained at assembly time.
check	Optional. If this specification is present (non-blank) the actual parameter will be tested for kind. Abnormal termination with a PARAMETER KIND ERROR diagnostic may occur at execution time.

Examples:

	VALUE 3,X1,,CHECK	Evaluates the third parameter into X1. Checks that the kind is correct.
	VALUE 5,X2,6	Evaluates the fifth parameter into X2 and assumes that it is a constant (6).
	SX3 1 VALUE X3,X3	Evaluates the first parameter into X3.
B	EQU 6 VALUE B,X6,8,C	Evaluates the sixth parameter into X6. Checks correctness of kind and assumes that it is a simple variable (8).

B.3.2. ASSIGN

This macro is applicable only to a parameter which is either a simple variable (8) or a subscripted variable (9) (i.e.,VS); and it optionally will check for these kinds. It assigns the contents of an X register to the specified parameter. Care must be taken to provide either a normalized floating point quantity or a boolean value in the X register. Use of the macro XFORM (See B.3.7) is advised for this purpose. Assignments will take place only into stack addresses in SCM; assigning into a subscripted variable in LCM not only will not produce the correct result, but it will destroy the contents of an SCM address.

ASSIGN	param,xreg,kind,check
param	{Absolute address expression, X register designator}. See B.3.1.
xreg	X-register designator. X register contains the value to be assigned to specified actual.
kind	Optional, absolute address expression. See B.3.1.
check	Optional. See B.3.1

Examples:

	SX1 XFORM ASSIGN	3 X1,X1,FR 5,X1,8,C	Assigns the value 3 (after converting it to floating point) to the fifth parameter. Assumes that it is a simple variable (8) and checks correctness of kind.
 N	MX2 SET ASSIGN	1 2 N,X2,,C	Assigns the boolean value true to the second parameter after checking that it is the correct kind.

B.3.3. ADDRESS

This macro is applicable to the following kinds of parameters only: a string (1), an array (5), a constant (6), a simple variable (8), a subscripted variable (9) or a virtual array (10), (i.e., GACVSR). Optionally, it will check for these kinds. It calculates the address, or first word address, of the specified parameter. The macro will also tell whether the computed address is in CM/SCM or in LCM.

Arrays and strings always grow from RA toward RA+FL, as opposed to the stack which grows from RA+FL toward RA.

ADDRESS	param,xreg,kind,check
param	{ Absolute address expression, X register designator } . See B.3.1.
xreg	X register designator. The X register will receive in the lower 18 bits the address† of a constant or a variable, or the first word address of an array or a string. Bit 59 will be 1 if the address is in LCM and 0 if in SCM (for arrays only).
kind	Optional. Absolute address expression. See B.3.1.
check	Optional. See B.3.1.

Examples:

ADDRESS	3,X1,5	Returns in X1 the fwa of the third parameter, assumed to be an array (5).
SX2 ADDRESS	4 X2,X3,,C	Returns in X3 the address of the fourth parameter and checks correctness of kind.

B.3.4. LENGTH

This macro is applicable to the following kinds of parameters: string (1), array (5), or virtual array (10). 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, X register designator } . See B.3.1.
xreg	X register designator. X register will receive the length of the actual parameter in machine integer form.
kind	Optional. Absolute address expression. See B.3.1.
check	Optional. See B.3.1.

†ALGOL 4.0 allows only 18 bit addresses for LCM

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:

Y	EQU LENGTH	3 Y,X2,1	Return in X2 the number of characters in the string (1) given by the third parameter.
	LENGTH	4,X4,10	Returns in X4 the length of the virtual array (10) given by the fourth parameter.
	SX1 LENGTH	1 X1,X7,,C	Returns the length of the array or string given by the first parameter after checking correctness of kind.

B.3.5. ORDER

This macro operates only on array (5) or virtual array (10) parameters. It returns the order, or dimensionality of the array (the number of subscripts required to address the array in its original declaration). Optionally, it checks that the kind is correct.

```
ORDER     param,xreg,check
```

param	{ Absolute address expression, X register designator }. See B.3.1.
xreg	X register designator. The X register will receive, in machine integer form, the order of the specified actual array parameter.
check	Optional. See B.3.1.

Examples:

```
ORDER     A, X2, CHECK
ORDER     X3,X7
```

B.3.6. 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, X register designator} . See B.3.1.
index	Optional. {Absolute address expression, X register designator} . If omitted, the 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 (machine integer form) must be the subscript of the required switch. The switch element if found after checking that the index is within the bounds of the switch (otherwise a fatal error will occur) and control is transferred accordingly.
check	Optional. See B.3.1. The specified actual will be checked to ascertain it is a label or a switch depending on the absence or presence of index.

Examples:

GOTO	3, , CCC	Checks whether the third parameter is a label and transfers control to it.
SX1 GOTO	5 X1,2	Transfers control to the second switch element of the switch given by the fifth parameter.

B.3.7. XFORM

This macro provides for the conversion of quantities into various data formats.

XFORM	operand, result, action
operand	X register designator. The X register contains the quantity to be transformed according to the action.
result	X register designator. The X register will receive the result of the conversion. The same register may be used as operand.
action	{ RI, RF,FR} . If RI, the operand is assumed to be a real and the result will be an integer in the ALGOL sense, i.e.: result:=entier (operand+0.5) If RF, the operand is a floating point quantity (real or integer) and the result will be a machine integer. If FR, the operand is a machine integer quantity and the result will be a floating point number.

Examples:

```
SX2       3           The contents of X3 will be the value 3 expressed as a normalized
XFORM     X2,X3,FR    floating point quantity.

XFORM     X6,X6,RI    The contents of X6 before the macro call are assumed to be a floating
                      point real quantity. After the macro call it will be a floating point
                      integer quantity.
```

B.3.8. STRING

This macro transforms a given COMPASS string into a string suitable for ALGOL and builds its descriptor. The resulting string may be passed as an actual parameter to an ALGOL procedure via the macro XEQ (See B.3.10).

```
STRING    name, string
```

name	A COMPASS symbol. This name will be used by the macro as the symbol to define the address where the descriptor will be placed, hence it must not be used anywhere in the location field in the COMPASS subprogram.
string	The string may contain any character from the character set, except the $. It may be formed by any number of characters, including zero (the empty string).

Usage of this macro will produce the non-fatal COMPASS error number 9 because it is not possible to change the micro marks (') of COMPASS, and they are used in ALGOL for opening and closing a string: '('THIS IS NOT A MICRO')'

Examples:

```
STRING   MYSTRG,(THIS STRING CONTAINS . , : ; [ ] )

STRING   A, (/,3B,-Z.DDDBDDD)

STRING   B            This string is empty
```

B.3.9. ERROR

This macro will print a given string and pass control to the routines which produce the traceback, according to the abnormal termination dump format option. (See 8.2.). Then the program will abort.

```
ERROR    string
```

string	The string may contain any character from the character set, except the $. The number of characters is limited to 135 and the empty string is not allowed.

Example:

```
ERROR    (THIS IS AN ERROR MESSAGE)
```

B.3.10. XEQ

This macro operates only on an actual parameter that is a no-type procedure with parameters (3) or without (13). The macro invokes this formal procedure from within the COMPASS subprogram. It permits the COMPASS subprogram to pass parameters to the formal procedure but they may be only simple variables (of types real, integer, or boolean), strings defined by the macro STRING, or alternatively other formals of the COMPASS subprogram. With the first type of parameter (3), the subprogram can transmit simple values to the formal procedure and receive results after execution. These parameters will be local to the COMPASS subprogram. With the last type of parameter (13), more complex kinds (arrays, switches, labels, procedures) can be transmitted from the original main program to the formal procedure via the COMPASS subprogram.

The XEQ macro always will produce code to check that the specified actual parameter is a no type procedure.

According to ALGOL up to 63 parameters could be passed to the formal procedure. The only limit imposed by the macro is that imposed by COMPASS, which allows up to 9 continuation lines per statement (See 5.4.3.).

```
XEQ   proc, (P1,P2, . . . , Pn)
```

proc	{Absolute address expression, X register designator} . The value, or contents, of proc is the ordinal of the no type procedure parameter. This parameter is referred to as the formal procedure.
P1, . . . , Pn	Optional. If omitted, the formal procedure is assumed to be a no-parameter procedure; and control will be given to it immediately. If given, Pi may have two forms:

param : typknd

param	{ Relocatable address expression, name given to a string } . If it is a relocatable address expression, param 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 it. If it is a name for a string use must have been made of the macro STRING.
:	Separator.
typknd	{ R,I,B,S } . The variable local to the COMPASS subprogram is of type real (R), integer (I) or boolean (B). For a string, S must be used.

*param

*	The asterisk means a special parameter.
param	{ Absolute address expression, X register designator } . The value or contents of param is the ordinal of the original parameter to the COMPASS subprogram which is to be transmitted to the formal procedure as its ith parameter.

Examples:

```
W   DATA    3.5
Y   VFD     1/1,59/0
Z   BSSZ    1
    XEQ     3, (W:R,Y:B,*5,Z:R)
```

Checks that the third parameter to the COMPASS subprogram is a no-type procedure. It transmits the real variable W, the boolean variable Y, the 5th parameter (to the COMPASS subprogram) and the real Z (where it expects a result) as the 1st, 2nd, 3rd, and 4th parameters respectively to the formal procedure given. Then the procedure is executed.

```
Q   EQU     10
    SX1     4
    SX2     6
    STRING  A,(/,6ZDD.DDD)
    XEQ     X1,(*Q,*X2,A:S)
```

Checks that the 4th parameter to the COMPASS subprogram is a no-type procedure. It transmits its 10th parameter, 6th parameter and the string named A as the 1st, 2nd, and 3rd parameters, respectively, to the formal procedure and executes it.

```
    XEQ     5
```

Checks that the 5th parameter to the COMPASS subprogram is a no-type procedure; and since there are no parameters, it executes it as a procedure without parameters.

B.4. STACK REQUEST MACRO (GETSCM)

This macro grants a given number of words in the local stack assigned to the COMPASS subprogram. The user requests this amount of core and provides a register where the fwa for his stack space will be found.

The stack grows from RA+FL towards RA; therefore, this area must be used by indexing the fwa negatively.

This macro must be used with extreme care, otherwise the stack and the code are liable to undetected destruction. Additionally, the space granted may be preset in accordance with the execution-time option S (See 8.3.).

```
GETSCM    size,fwa
```

size	{ X register designator, absolute address expression } . The contents or value of size indicate the number of words to be requested.
fwa	X register designator. This X register will have the fwa of the granted stack space. It may be the same register as size. This area comprises from fwa to fwa-size+1 (from RA+FL towards RA).

Examples:

```
     SX5        7
     GETSCM     X5,X6     In both cases 7 words of stack are requested; and when granted, the fwa will be
     GETSCM     7,X6      found in X6.
```

B.5. EXAMPLES

In the following examples, both the calling ALGOL program and the COMPASS subprogram are given.

To assemble the COMPASS subprogram, the control card must be:

```
     COMPASS(S=ALGTEXT)
```

B.5.1 EXAMPLE 1

The calling ALGOL program is:

```
'BEGIN' 'COMMENT' THIS PROGRAM WILL PRINT THE VALUES 1   2   3   4   IF THE COMPASS
                  PROCEDURE JUMP IS CORRECT, OTHERWISE VALUES LIKE 300, 400, 500 AND
                  1000 WILL APPEAR;

'SWITCH' S:=L5,L4,L3;
'PROCEDURE' JUMP(A,B,C);
              'LABEL' A; 'SWITCH' B; 'BOOLEAN' C; 'CODE' 100;
                  OUTREAL(61,1);
                  JUMP(L1,S, 'TRUE');
                  OUTREAL(61,300);
              L1: OUTREAL(61,2);
                  JUMP(L2,S,'FALSE');
              L5: OUTREAL(61,500);
                  'GOTO' END;
              L4: OUTREAL(61,400);
                  'GOTO' END;
              L2: OUTREAL(61,1000);
                  'GOTO' END;
              L3: OUTREAL(61,3);
            END:  OUTREAL(61,4)
'END'
```

The COMPASS subprogram is:

```
                  IDENT JUMPS
                  SST

*THIS PROGRAM TESTS THE 3RD PARAMETER (A BOOLEAN). IF IT IS 'TRUE' WILL GIVE
*CONTROL TO THE LABEL (1ST PARAMETER). IF IT IS 'FALSE' WILL GIVE CONTROL TO
*THE THIRD ELEMENT IN THE SWITCH LIST SPECIFIED BY THE 2ND PARAMETER
```

```
                ALGOL    100
                SPEC     (1:L,2:W,3:CV)
                VALUE    3,X7
                PL       X7,FALSE
                GOTO     1                    .A LABEL
                JP       FIN
         FALSE  GOTO     2,3                  .A SWITCH
         FIN    RETURN
                END
```

B.5.2. EXAMPLE 2

The calling ALGOL program is:

```
'BEGIN' 'COMMENT' THIS PROGRAM WILL OUTPUT THE 10 ELEMENTS OF THE ARRAY R, WHERE
                  THE COMPASS PROCEDURE TEST HAS STORED THE FOLLOWING ITEMS:
                  NUMBER OF PARAMETERS IN THE CALL (8), KIND OF A (0), KIND OF D (3),
                  KIND OF G (6), TYPE OF B (0), TYPE OF E (2), TYPE OF H (2), LENGTH OF B
                  (10), LENGTH OF F (10), ORDER OF F (1).
                  FINALLY Q IS ALSO OUTPUT (100);
'REAL' P;
'PROCEDURE' ONE(A);'REAL' A; A:=A;
'REAL' 'PROCEDURE' TWO(A); 'REAL' A; TWO:=A;
'PROCEDURE' TEST (A,B,C,D,E,F,G,H);
            'SWITCH' A; 'STRING' B; 'LABEL' C;
            'PROCEDURE' D; 'REAL''PROCEDURE' E; 'ARRAY' F; 'REAL' G,H;
            'CODE';
'SWITCH' S:=L1,L2;
'ARRAY' R[1:10];
'REAL' Q;
L1:TEST(S,'('TEST')',L2,ONE,TWO,R,100,Q);
L2:OUTARRAY(61,R);
   OUTREAL(61,Q);
'END'
```

The COMPASS subprogram is:

```
          IDENT       PASSPAR
          SST

*THE FOLLOWING MACRO STORES A REAL QUANTITY IN SUCCESSIVE ARRAY ELEMENTS

  STORE   MACRO
          XFORM       X7,X6,FR          PUT IN X6 THE FLOATING POINT VALUE OF
*                                       THE FIXED POINT QUANTITY ORIGINALLY IN X7
          SA1         FWAARR
          SA6         X1
          SX7         X1+1
          SA7         FWAARR            .STORE INTO ARRAY
          ENDM
```

```
TEST     ALGOL
         SPEC       (1:W,2:G,3:L,4:PFNT,5:FT,6:A,7:CXVST,8:VS)
         ADDRESS    6,X6,5
         SA6        FWAARR              .STORE FWA FOR ARRAY
         PARAMS     X7
         STORE
         KIND       1,X7
         STORE
         KIND       4,X7
         STORE
         KIND       7,X7
         STORE
         TYPE       2,X7
         STORE
         TYPE       5,X7
         STORE
         TYPE       8,X7
         STORE
         VALUE      7,X6,6
         ASSIGN     8,X6
         LENGTH     2,X7,1
         STORE
         LENGTH     6,X7,5
         STORE
         ORDER      6,X7
         STORE
         RETURN
FWAARR   DATA       0
         END
```

CHARACTER REPRESENTATION OF ALGOL SYMBOLS C

Table 1. Character Representation of ALGOL Symbols

ALGOL Symbol	48-Character Representation	Additional Representation	ALGOL Symbol	48-Character Representation
A-Z	A-Z		true	'TRUE'
a-z	~		false	'FALSE'
0-9	0-9		go to	'GO TO'
+	+		if	'IF'
-	-		then	'THEN'
x	*		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'
⊔	⊔		algol††	'ALGOL'
(	(		eop††	'EOP'
:=	.= or ..=	:=		
)	)			
[	(/	[		
]	/)	]		
‘	'('			
’	')'			

† In a format string, must be represented by an asterisk.

†† Not defined in the ALGOL-60 Revised Report; code and algol are defined in Section 5.4.1., Chapter 2; eop in Chapter 4.

ALGOL 4.0 allows two forms of character representation of ALGOL symbols: the ALGOL Extended 62-character set and the 48-character subset formerly required by an earlier CDC ALGOL compiler. 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 following sample program is in the form that is punched into cards according to the 48-character subset:

```
TWO-DIMENSIONAL ARRAY: 'BEGIN' 'INTEGER' I.,
'COMMENT' 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.,
  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'
```

The same program punched according to the ALGOL Extended 62-character set would look as follows:

```
TWO-DIMENSIONAL ARRAY: 'BEGIN' 'INTEGER' I;
'COMMENT' 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;

    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'
```

STANDARD PROCEDURES FOR VECTOR AND MATRIX MANIPULATIONS D

A set of built-in procedures will be provided for certain operations on arrays. These procedures take full advantage of the characteristics of the CDC CYBER 70 models 74 and 76 instruction stack. Each procedure is mainly equivalent to ALGOL for statements as will be shown in the definitions of these procedures.

The provision of the array procedures invests the computer with pipelined functional streaming units at the level of the source language. By defining the machine code for the macro expansions so that it loops wholly within the instruction stack, an emulation of pipelining is achieved with very fast execution speed. Only those array procedures whose machine code can so fit within the instruction stack are provided. There is no general user macro definition facility at the source language level at present.

The use of these procedures will produce execution speed gains on all machine models due to the compacting of the code. However, such gains are more significant for Models 74 and 76, where they can be very appreciable. In all cases, a vector must be of length 10 or greater before gains are appreciable (see Chapter 12, section 12.3).

Except for initialization, all the procedures will execute within the instruction stack of both Model 76 and Model 74.

At present, only a preliminary list of the procedures can be given. The list could be extended.

D.1 MATRIX AND VECTOR PROCEDURES

Twelve procedures are added to the standard procedure set. Their designs are given here in terms of ALGOL procedures, but they do not imply that they are implemented as such.

Note: in the examples,upb and lwb are used to represent the upper and lower bounds of an array. They do not represent ALGOL procedures and are only used to facilitate the explanation.

D.1.1 MATRIX PROCEDURE

MATMULT: The standard procedure MATMULT does the crossed product of two matrices M1 (m,n) by M2 (n,p) giving the matrix M3 (m,p). The bound pairs are assumed to be compatible, else a run-time error occurs. And arrays M1, M2 and M3 must have 2 dimensions.

```
procedure MATMULT (M1,M2,M3,M,N,P); value M,N,P; integer M,N,P; array M1,M2,M3;
    begin integer I,J,K,L1,L2,L3,R1,R2,R3;
        real S;

            if M< 1 v N< 1 v P<1 then ERROR ('MATMULT - NEGATIVE DIMENSIONS');

            L1: = lwb (M1,1);      R1:=lwb (M1,2);
            L2: = lwb (M2,1);      R2:=lwb (M2,2);
            L3: = lwb (M3,1);      R3:=lwb (M3,2);
    if    ( (upb (M3,1) - L3+1) ≠M)
        ∨ ( (upb (M1,2) - R1+1) ≠N)
        ∨ ( (upb (M2,2) - R2+1) ≠P) then ERROR   ('MATMULT - INCONSISTANT MATRIX SIZES');

    for   I:=0 step 1 until M-1 do
          for   J:=0 step 1 until P-1 do
                      begin  S:=0;
                            for   K:=0 step 1 until N-1 do
                                  S:=S + M1 [L1+I, R1+K]  *M2 [L2+K, R2+J] ;
                                        M3 [L2+I, R2+J] :=S
                      end
end MATMULT
```

D.1.2 VECTOR PROCEDURES

Twelve procedures are concerned with vectors. By vector is meant a one-dimensional array, the type of which is either boolean or real (but they can't be mixed).

Their designs are given, in terms of ALGOL procedure, below. Here is some information, in the case where lower-bound = 0 and upper-bound = N for each vector:

VINIT (OP,E,S,V1)

build constant vector:	OP ⩽ 1; V1 [I] :=E, I from 0 to N
arithmetic progression:	OP = 2; V1 [0] :=E; V1 [I] := V1 [I-1] +S, I from 1 to N
geometric progression:	OP ⩾ 3; V1 [0] :=E; V1 [I] := V1 [I-1] *S, I from 1 to N

VXMIT (OP, V1, V2, INIT, STEP);

OP ⩽ 1 transfers V1 to part of V2: V2 [INIT+i*STEP] :=V1 [i]

OP ⩾ 2 transfers part of V2 to V1: V1 [i] := V2 [INIT+i*STEP]

VXMIT is a function that transfers a row or a column of a matrix to or from a vector.

VMONOD (OP,V1, V2)

V2 [i] :=OP V1 [I] where OP is a monadic operation as follows:

OP	⩽ 1	transmit	
	= 2	reversed transmit	
	= 3	real to integer transform	
	= 4	negate	
	= 5	absolute	
	= 6	entier (floor)	
	= 7	ceiling	
	= 8	sign	
	= 9	delta	V2 [i] := V1 [i+1] - V1 [i]
	⩾ 10	mean	V2 [i] := (V1 [i+1] + V1 [i])/2

VDIAD (OP, V1,V2,V3)

V3[i] := V1[i] OP V2[i] where OP is a diadic operation as follows:

OP	≤ 1	add	
	$= 2$	subtract	
	$= 3$	multiply	
	$= 4$	divide	
	$= 5$	average	V3[i] :=(V1 [i] + V2[i])/2
	≥ 6	average difference	V3 [i] := (V1 [i] - V2 [i])/2

VSDIAD (OP, E, V2, V3)

V3[i] := E OP V2[i] where E is a (broadcasted) real expression and OP as for VDIAD. E is evaluated only once, on entry to the procedure.

VPROSUM (OP, V1)

sigma:	VPROSUM:= V1[0] + V1[1] + . . . + V1[N] when OP $\leq$1
product:	V1[0] * V1[1] * . . . * V1[N] when OP $\geq$2

VDOT (V1,V2)

dot product: VDOT:= V1[0] * V2[0]+ V1[1] * V2[1]+ . . . +V1[N] *V2[N]

BNOT (S1,S2)

monadic operation: S2[i] := ¬S1[i]

VBOOL (OP, S1,S2,S3)

S3[I] := S1[I] OP S2[I] where OP is a diadic operation as follows:

OP	≤ 1	or	OP = 7	pierce
	$= 2$	and	≥ 8	inhibit
	$= 3$	imply		
	$= 4$	equivalence		
	$= 5$	exclusive or		
	$= 6$	stroke		

The truth table for boolean operations is (0 stands for false and 1 for true):

Source		$\vee$	$\wedge$	$\supset$	$\equiv$	Excl. or	Stroke	Pierce	Inhibit
X	Y								
0	0	0	0	1	1	0	1	1	0
0	1	1	0	1	0	1	1	0	0
1	0	1	0	0	0	1	1	0	1
1	1	1	1	1	1	0	0	0	0

VSBOOL (OP, B, S2,S3)

S3[i] := B OP S2[i] where B is a (broadcasted) boolean expression and OP as in VBOOL. E is evaluated only once, on entry to the procedure.

Note: VSBOOL (1,true,V1,V1) sets each element of V1 to true
VSBOOL (2,false,V1,V1) sets each element of V1 to false
VSBOOL (2,true, V1,V2) copies V1 into V2

VREL (OP, V1, V2, S3)

compare element by element vectors V1 and V2:

S3[i] := V1[i] OP V2[i] where OP is an operation as follows:

OP	$\leqslant 1$	$<$
	$= 2$	$\leqslant$
	$= 3$	$=$
	$= 4$	$\geqslant$
	$= 5$	$>$
	$\geqslant 6$	$\neq$

Furthermore, VREL takes the value true if and only if every element of the result vector S3 is true.

VSREL (OP,E,V2,S3)

as in VREL but E is a broadcasted real expression. E is evaluated only once, on entry to the procedure.

The corresponding procedures are designed in such a way that the length of the result vector has priority on other vectors lengths: if necessary, operand vectors are truncated or filled with zeroes or false.

```
procedure VINIT (OP,E,S,V1);value OP,E,S;
          integer OP; real E,S;array V1;
     begin integer I;
          if OP ≤ 1 then begin for I:=lwb(V1,1) step 1 until upb (V1,1)
                                             do V1[I] :=E
                           end
                       else
          if OP = 2 then begin V1[lwb(V1,1)] :=S;
                                     for I:=lwb(V1,1+1 step 1 until upb(V1,1)
                                        do V1[I]:=V1[I-1]+E
                          end
                       else
          if OP ≥ 3 then begin V1[lwb(V1,1)] :=S;
                                     for I:=lwb(V1,1)+1 step 1 until upb(V1, 1)
                                        do V1[I] :=V1[I-1]*E
                           end
end VINIT
```

```
procedure VXMIT (OP,V1,V2,INIT,STEP); value OP, INIT, STEP;
          array V1,V2; integer OP, INIT, STEP;
     begin integer I,J,L1,L2 ;
               L1:= 1wb (V1,1); U1:=upb (V1,1)-L1;
               L2:= 1wb (V2,1); U2:=upb (V2,1)-L2;
          if OP ≤1 then
               begin for I:= 0 step 1 until U1 do
                    begin J:= INIT+I*STEP;
                         if J ≥0 ∧ J≤U2 then V2[L2+J] :=V1[L1+I]
                    end
               end
```

```
                  else for I:=0 step 1 until U1 do
                              begin J:= INIT+I*STEP;
                                   V1[L1+I] :=if J ⩾ 0 ∧ J ⩽ U2 then V2[L2+J] else 0
                              end
end VXMIT;

procedure VMONOD (OP, V1,V2); value OP; integer OP; array V1,V2;
     begin integer L1,U1,L2,U2,L,I;
          switch ACTION:=XMIT, REVERS,XFORM,NEGATE,ABSOLUTE,FLOOR,CEILING,SIGNE,
                          DELTA,MEAN;

          L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
          L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
             L:=if U2 ⩽ U1 then U2 else U1;
             goto ACTION [if OP ⩽ 1 then 1 else if OP ⩾ 10 then 10 else OP];

XMIT:  OP equals 1:
      for I:=0 step 1 until L do V2[L2+I] :=V1[L1+I];
      goto FILLING;

REVERS:  OP equals 2:
          for I:=0 step 1 until L do V2[L2+I] :=V1 [U1+L1-I];
          goto FILLING;

XFORM:   OP equals 3:
          for I:=0 step 1 until L do V2[L2+I] :=entier (V1[L1+I]+.5);
          goto FILLING;

NEGATE:  OP equals 4:
          for I:=0 step 1 until L do V2[L2+I] := -V1[L1+I];
          goto FILLING;

ABSOLUTE:  OP equals 5:
            for I:=0 step 1 until L do V2[L2+I] := abs(V1[L1+I]);
            goto FILLING;

FLOOR:   OP equals 6:
          for I:=0 step 1 until L do V2[L2+I] :=entier (V1[L1+I]);
          goto FILLING;

CEILING:  OP equals 7:
          for I:=0 step 1 until L do V2[L2+I] := if  V1[L1+I] =entier(V1 [L1+1])
                                            then  V1[L1+I] else  entier(V1[L1+I]+1);
          goto FILLING;

SIGNE:    OP equals 8:
          for I:=0 step 1 until L do V2[L2+1] :=sign (V1[L1+I]);
          goto FILLING;
      DELTA:   OP equals 9:
               for I:=0 step 1 until L-1 do
                     V2[L2+I] := V1[L2+I+1] - V1[L1+I];
                     V2[L2+L] := (if L+1 ≤ U1 then V1 [L1+L+1] else 0) - V1[L1+L];
               goto FILLING;

      MEAN:    OP equals 10:
               for I:=0 step 1 until L-1 do
                     V2[L2+I] := (V1[L2+I+1] - V1[L1+I])/2;
                     V2[L2+L] :=( (if L+1 ≤U1 then V1[L1+L+1] else 0)-V1[L1+L])/2;

      FILLING: Fill V1 with zeroes:
               for I:=L+1 step 1 until U2 do V2[L2+I] :=0

end VMONOD;

procedure VDIAD (OP,V1,V2,V3); value OP; integer OP; array V1,V2,V3;

      begin integer L1,U1,L2,U2,L3,U3,I;
            real procedure action (OP,X,Y); value OP,X,Y;
                            integer OP; real X,Y;
                  action:=   if OP ≤ 1 then X+Y else
                             if OP = 2 then X-Y else
                             if OP = 3 then X*Y else
                             if OP = 4 then X/Y else
                             if OP = 5 then (X+Y)/2
                                   else       (X-Y)/2;
            L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
            L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
            L3:=1wb (V3,1); U3:=upb (V3,1)-L3;

            for I:=0 step 1 until U3 do
                  V3[L3+I] :=action (OP, if I ≤U1 then V1[L1+I] else 0,
                                         if I ≤U2 then V2[L2+I] else 0)
end VDIAD;

procedure VSDIAD (OP,E,V2,V3); value OP,E;
            integer OP; real E; array V2,V3;

      begin integer L2,U2,L3,U3,I;
            real procedure action (OP,X,Y) value OP,X,Y;
                  integer OP; real X,Y;
            action:= if OP ≤ 1 then X+Y else
                     if OP = 2 then X-Y else
                     if OP =3 then X*Y
                       else        X/Y;
            L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
            L3:=1wb (V3,1); U3:=upb (V3,1)-L3;
            for I:=0 step 1 until U3 do
                  V3[L3+I] :=action (OP,E, if I ≤ U2 then V2[L2+I] else 0)
end VSDIAD;
```

```
real procedure VPROSUM (OP,V1); value OP;
          integer OP, array V1;

        begin integer I,LB;real P;
              LB:=lub(V1,1);P:=V1 [LB];
              for I:=LB+1 step 1 until upb (V1,1) do
              P:= if OP ⩽ 1 then P+V1 [I]
                          else P*V1 [I];
              VPROSUM:=P
end VPROSUM;

real procedure VDOT (V1,V2); array V1,V2;

    begin integer L1,U1,L2,U2,L,I;
        real R;
        R:=0;
        L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
        L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
        L:= if U1 ⩽ U2 then U1 else U2;
        for I:=0 step 1 until L do R:=R+V1 [L1+I] *V2 [L2+I];
        VDOT:=R
end VDOT;

procedure BNOT (S1,S2); boolean array S1,S2;

    begin integer L1,U1,L2,U2,L,I;
        L1:=1wb (S1,1); U1=upb (S1,1)-L1;
        L2:=1wb (S2,1); U2:=upb (S2,1)-L2;
        L:=if U1 ⩽ U2 then U1 else U2;
        for I:=0 step 1 until L do S2[L2+I] :=¬S1 [L1+I];
        for I:=L+1 step 1 until U2 do S2 [L2+1] :=false
end BNOT;

procedure VBOOL (OP,S1,S2,S3); value OP;
        integer OP; boolean array S1,S2,S3;

    begin integer L1,U1,L2,U2,L3,U3,I;
        boolean procedure action (OP,X,Y); value OP,X,Y;
              integer OP; boolean X,Y;
        action:=if OP ⩽ 1 then X∨Y else
                 if OP = 2 then X∧Y else
                 if OP = 3 then X⊃Y else
                 if OP = 4 then X≡Y else
                 if OP = 5 then ¬(X≡Y) else
                 if OP = 6 then ¬(X∧Y) else
                 if OP = 7 then ¬(X∨Y)
                    else        X∧¬Y;
```

```
            L1:=lwb (S1,1); U1:=upb (S1,1)-L1;
            L2:=lwb (S2,1); U2:=upb (S2,1)-L2;
            L3:=lwb (S3,1); U3:=upb (S3,1)-L3;
            for I:=0 step 1 until U3 do
                  S3[L3+I] :=action (OP, if I ≤ U1 then S1[L1+I] else false,
                                         if I ≤ U2 then S2[L2+I] else false)
end VBOOL;

procedure VSBOOL (OP,B,S2,S3); value OP,B; integer OP; boolean B; boolean array S2,S3;
      begin integer L2,U2,L3,U3,I;
            boolean procedure action (OP,X,Y); value OP,X,Y;
                  integer OP; boolean X,Y;
            action:= if OP ≤ 1 then X∨Y else
                     if OP = 2 then X∧Y else
                     if OP = 3 then X⊃Y else
                     if OP = 4 then X≡Y else
                     if OP = 5 then ¬(X≡Y) else
                     if OP = 6 then ¬(X∧Y) else
                     if OP = 7 then ¬(X∨Y)
                        else        X∧¬Y;

            L2:=lwb (S2,1); U2:=upb (S2,1)-L2;
            L3:=lwb (S3,1); U3:=upb (S3,1)-L3;
            for I:=0 step 1 until U3 do
                  S3[L3+I] :=action (OP,B,if I ≤ U2 then S2[L2+I] else false)
end VSBOOL;

boolean procedure VREL (OP,V1,V2,S3); value OP; integer OP; array V1,V2; boolean array S3;
            begin integer L1,U1,L2,U2,L3,U3,I; boolean B;
                  boolean procedure action (OP,X,Y); value OP,X,Y; integer OP; real X,Y;
                  action:=if OP ≤ 1 then X < Y else
                          if OP = 2 then X ≤ Y else
                          if OP = 3 then X = Y else
                          if OP = 4 then X ≥ Y else
                          if OP = 5 then X > Y
                             else        X ≠ Y;
                  B:=true;
                  L1:=lwb(V1,1);U1:=upb(V1,1)-L1;
                  L2:=lwb(V2,1);U2:=upb(V2,1)-L2;
                  L3:=lwb(S3,1);U3:=upb(S3,1)-L3;
                  for I:=0 step 1 until U3 do
                        begin
                          S3[L3+I] :=action (OP, if I ≤ U1 then V1[L1+I] else 0,
                                                 if I ≤ U2 then V2[L2+I] else 0)
                          B:=B∧S3[L3+I];
                          end;
                          VREL:=B
end VREL;
```

```
boolean procedure VSREL (OP,E,V2,S3); value OP,E; integer OP; real E; array V2; boolean array S3;

    begin integer L2,U2,L3,U3,I; boolean B;
        boolean procedure action (OP,X,Y); value OP,X,Y; integer OP; real X,Y;
        action:=if OP ≤ 1 then X < Y else
                if OP = 2 then X ≤ Y else
                if OP = 3 then X = Y else
                if OP = 4 then X ≥ Y else
                if OP = 5 then X > Y
                   else        X ≠ Y;

       L2:=lwb(V2,1);U2:=upb(V2,1)-L2;
       L3:=lwb(S3,1);U3:=upb(S3,1)-L3;
       B:=true;

        for I:=0 step 1 until U3 do
           begin
             S3 [L3+I] :=action (OP,E if 1≤U2 then V2 [L2+I]  else 0)
             B:=B∧S3 [L3+I] ;
           end;
        end VSREL;
```

Note: Due to the CYBER–ALGOL implementation of arrays (row by row), the eleven vector procedures may be used by replacing one or more one-dimension arrays by n-dimensional arrays (of the same kind).

Example 1:

```
array M[1:3,2:4] ; M[1,2] :=1;
VINIT (2,1,M)
L: . . .
```

when control reaches the label L, M is filled as follows:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Example 2:

```
array V,W[-4:4] ,M,N [1:3,2:4] ;
W[-4] := 1;
VINIT (1,1,V); VINIT(2,1,W);
VDIAD(1,V,W,M); VMONOD(2,M,N);
L:
```

when control reaches the label L, N is filled as follows:

$$\begin{pmatrix} 10 & 9 & 8 \\ 7 & 6 & 5 \\ 4 & 3 & 2 \end{pmatrix}$$

Example 3:

```
array C[1:N,1:N], T[1:M,1:P], VM[1:M], VN[1:N-1], VP[1:P];
....
FETCH UPPER SUB DIAGONAL OF C:
        VXMIT(2,VN,C,2,N+1);
SEND 0 INTO EACH ELEMENT OF ROW 4 OF T:
        VINIT(1,0,0,VP);
        VXMIT(1,VP,T,4,M);
...
```

INDEX

COMMENT SHEET

CONTROL DATA
CORPORATION

TITLE: ALGOL 4

PUBLICATION NO. 60384700 REVISION A

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).

CUT ON THIS LINE

General comments:

FROM NAME: ____________________ POSITION: ____________________

COMPANY NAME: ____

ADDRESS: ____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE STAPLE

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

CUT ON THIS LINE

FOLD FOLD

STAPLE STAPLE

CONTROL DATA

1/2
3/4
1
1-1/4

►►CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

Pub. No. 60384700

___ Litho in U.S.A.