

Dartmouth ALGOL

for the GE-265
Time-Sharing System

GENERAL  ELECTRIC

DARTMOUTH ALGOL
for the
GE-265 TIME-SHARING SYSTEM

June 1965

Rev. February 1966

GENERAL  **ELECTRIC**

COMPUTER DEPARTMENT

PREFACE

This edition incorporates the latest additions and changes to ALGOL as used in the computer time-sharing service provided by General Electric. A vertical bar in the margin identifies the changes. Section I, "Error Messages", has been added in this edition. The previous edition may still be used.

Initial printing	June 1965
Revised	February 1966

TABLE OF CONTENTS

	Page
A. Introduction	1
B. Transliteration	1
C. Input	2
D. Data	2
E. Output	3
F. Standard Functions	3
G. Restrictions	4
H. Anomalies and Limitations	5
I. Error Messages	6
J. Guides to ALGOL 60	11

APPENDIX

Revised Report on the Algorithmic Language ALGOL 60	13
---	----

A. INTRODUCTION

A fairly complete version of ALGOL 60 is available through General Electrics Computer Time-Sharing Service. The version is called "Dartmouth ALGOL" because it, like the Time-Sharing System, was developed at Dartmouth College.

This booklet describes how Dartmouth ALGOL differs from ALGOL 60. It presumes a knowledge of ALGOL 60 such as may be obtained from any of the guides to the ALGOL 60 language. A list of some of these guides has been appended. For reference, the official ALGOL 60 report also is appended.

To use Dartmouth ALGOL with the Computer Time-Sharing Service also requires a knowledge of how to use the Time-Sharing System Commands. Refer to the Time-Sharing System Manual, CPB-1182, for this information. The manual is available from any of the Information Processing Centers listed on the back cover.

B. TRANSLITERATION

1. The ALGOL 60 symbol indicated on the left below is typed on the Teletypewriter as indicated on the right.

<u>ALGOL 60</u>	<u>Dartmouth ALGOL (Teletypewriter)</u>
multiplied by tens exponent	*
\div	\$
\uparrow	shift L
\wedge	NOT
\vee	AND
\cup	OR
\equiv	IMPLY
go to	EQUIV
{	GOTO or GO space TO
}	shift K K
\leq	shift M M
\geq	< =
\neq	> =
,	=/ or /=
	"

Whenever the transliteration consists of more than one character, the characters must not be separated by spaces--except for GO space TO (GO TO) where, as indicated, a single space is allowed.

2. Program text may be typed in a format independent of the Teletypewriter line except that each line must begin with a non-numeric, so that the sequence number on each line can be distinguished, and that the carriage return at the end of each line, required to enter the line, appears in the program text as a space.

C. INPUT

Input is accomplished by means of a call to a procedure named READATA. The call parameters are: (1) a data name, and (2) a list of names of variables given in the order in which the data are to be assigned.

Example: READATA (ELEMENTS, A, B [I, J] , C [J+2] , D);

If more names are listed than there are values given in data declarations for the named data, an error message to that effect is given during execution. If there are fewer names than values, unassigned values are available for subsequently executed READATA calls with the same data name.

No match between the types of variables listed in the call and the types of values provided in the data is performed.

D. DATA

Data may be provided in two ways. In the first to be described, the data is provided with the program; in the second, it is provided in the course of execution.

1. When data is provided with the program, it is written in a declaration introduced by the word DATA. Like a switch declaration, a data declaration consists of a name followed by a colon; equal sign followed by a list. In the case of the data declaration, the name is a data name, and the data list consists of real, integer, or Boolean values.

Example: DATA ELEMENTS: = 14.3\$2, -15.2, 5, TRUE;

Unlike the switch declaration, though, the data declaration may appear anywhere within the block.

The data name must occur in the same block as the declaration for it and may appear only in two contexts. It may be the first parameter in a call to the READATA procedure, or it may be an actual parameter in a call to a procedure having a matching formal parameter specified to be a data name. The word DATA is used as a specifier for this purpose, and data may not be called by value.

Example: PROCEDURE P(DN, A, B); DATA DN; REAL A, B;
READATA (DN, A, B);
P(ELEMENTS, FIRST, SECOND);
DATA ELEMENTS: =.30103, 25;

A RESTORE statement causes the subsequent READATA statements to read the data list again, beginning with the first item. The block name is specified in parentheses after the word RESTORE.

Example: 10 BEGIN REAL A, B, C;
20 READATA (BLOCK, A, B, C);
30 PRINT (A, B, C);
40 READATA (BLOCK, A, B, C);
50 PRINT (A, B, C);
60 RESTORE (BLOCK);
70 READATA (BLOCK, A, B, C);
80 PRINT (A, B, C);
81 DATA BLOCK: =1, 2, 3,
82 4, 5, 6; END;
90 END
RUN

The resulting printout would be

```
1    2    3
4    5    6
1    2    3
```

2. It is possible to generate a "call for input" within an ALGOL program. This is done by using a normal READATA procedure call with TELETYPE as the list name. An example is:

```
READATA(TELETYPE, X, Y, Z);
```

This statement would cause a ? to appear on the user's Teletype, and the system would then wait for him to input a line of data and give a carriage return. The only legal input at present is a string of legal numbers equaling the number called for in the READATA statement. For instance, as input to the previous example, one might type:

```
123456, 12$60, -123.678 (carriage return)
```

In general, anything legal in a data list within an ALGOL program is legal within an input statement. The only additional restriction is that the number of data elements given must match the number called for in the READATA statement.

E. OUTPUT

Output is accomplished by means of a call to a procedure named PRINT. The parameters are given in a list of names of variables or expressions the values of which are to be output or of values or literals which are themselves to be output.

Literals are indicated by enclosing them within (double) quotation marks.

```
Example: PRINT (A, B [I, J+1] , 3, "SCORE=", "'", V+A*B)
```

An empty literal, that is, one without any characters between quotation marks, may be used to indicate that normal spacing between the output of the left-hand and right-hand values is to be suppressed.

All of the output requested in one call is typed on an integral number of lines. If an empty literal is the last in the print list, the carriage return at the end of the print line is suppressed.

F. STANDARD FUNCTIONS

The identifier displayed on the ~~right~~^{left} is used to indicate a call to a function which returns the value described on the right. In this description, V indicates the value of the parameter with which the function is called.

ABS	absolute value of V
SIGN	+1 if V > 0 0 if V = 0 -1 if V < 0
SQRT	square root of V
SIN	sine of V given in radians

COS	cosine of V given in radians
ARCTAN	principal value in radians of arc tangent of V
LN	natural logarithm of V
EXP	Eulerian constant raised to the V power
ENTIER	largest integer not greater than V
RANDOM	next in a sequence of (pseudo) random numbers rectangularly distributed in the range $0 \leq \text{RANDOM} < 1$

A new procedure has been added to the ALGOL library. It is a no argument procedure called TIME and gives the number of seconds that your program has been running. An example of its use is:

```
IF TIME > LIMIT THEN GOTO CRUMP;
```

It gives the user a real-time check on the amount of computing time his program is taking.

G. RESTRICTIONS

All of the facilities of ALGOL 60 are provided in Dartmouth ALGOL except as noted below:

1. Procedure definition required before reference. Reference to a procedure (either a call to or a mention of as an actual parameter) is allowed only in text lexically after the declaration of the procedure.
2. Recursive procedure call not allowed. Most of the kinds of calling referred to as recursive are prohibited. A call to a procedure within that procedure is not allowed. Restriction 1, above, also prohibits a call within one procedure to another which calls the original (since the other would have to be declared before the original and the original before the other).

Nested calls to a function such as "REAL PROCEDURE F(A); REAL A; F: = A+2; PRINT(F(F(2)));" are not allowed. When the function is a standard function, however, nested calling, such as "LN(LN(25))" is permissible.

3. Specification of formal parameters is mandatory. The specification of formal parameters which is optional in ALGOL 60 (except for parameters specified to be called by value) is mandatory in Dartmouth ALGOL.

Declarations of variable types in procedures need not match the variable types in the procedure call. i. e., the following is legal as a program:

```
BEGIN INTEGER NO1, NO2; REAL NO3, NO4;
PROCEDURE EXAMPLE (VAR1, VAR2); REAL VAR1; INTEGER VAR2;
    BEGIN END;
    EXAMPLE (NO1, NO3);
END
```

4. Constant array declarations. Lower and upper bounds for subscripts for an array must be declared as numeric constants. The OWN ARRAY declaration is not allowed.

5. Call by value arrays or procedures are not allowed. A formal parameter specified to be an array or (parameter-less function) procedure may not also be specified to be called by value.
6. Restrictions on identifiers.
 - Identifiers may not contain lower case letters.
 - Identifiers may contain embedded blanks but these are ignored; hence "A B", "A B", an "AB" are the same.
 - Identifiers may not contain more than 30 characters (not counting blanks, of course).
 - Identifiers the same as the basic symbols of Dartmouth ALGOL are not allowed. Moreover, identifiers with embedded blanks must not consist of a basic symbol followed by one or more blanks. The following are the basic symbols that may occur as identifiers (or words of identifiers) in ALGOL 60 that are not allowed in Dartmouth ALGOL.

AND	END	LABEL	SWITCH
ARRAY	EQUIV	NOT	THEN
BEGIN	FALSE	OR	TRUE
BOOLEAN	FOR	OWN	UNTIL
COMMENT	GOTO	PROCEDURE	VALUE
DATA	IF	REAL	WHILE
DO	IMPLY	STEP	
ELSE	INTEGER	STRING	

- Identifiers the same as the name of a standard function or procedure (READATA or PRINT) or data name (TTY) may not be used for other identification.
7. Strings are legal in procedure calls. They must be declared as such as follows:

```
PROCEDURE EXAMPLE2 ( X, Y ); REAL X; STRING Y;
      BEGIN PRINT (Y); END;
EXAMPLE 2 (A, "STRING CAN BE ANYTHING IN QUOTES");
```

(Obviously some of the above program is missing.)

8. Other restrictions.
 - Unsigned integer labels are not provided.
 - The elaborate parameter delimiters are not provided.

H. ANOMALIES AND LIMITATIONS

There are two unannounced inabilities of the Dartmouth ALGOL that have been noted. Until corrected these anomalies should be avoided.

1. Arithmetic relations in a conditional BOOLEAN expression are not allowed. A conditionally BOOLEAN expression as a constituent of a relation in the IF clause.

Example: REAL R; BOOLEAN B; B:= IF R> 1 THEN TRUE ELSE FALSE;
(cf McCracken, p. 47)

This is regarded as erroneous, however, by the Dartmouth ALGOL.

2. Label must be so identified before being used as an actual parameter. When a statement label is used as an actual call parameter, it must have been declared in text lexically prior or have been included in the switch list of a switch declaration in the block head.
3. Number and dimension limitations. Dartmouth ALGOL is limited to:
 - integers within the range $\pm 2^{38}$ (about 11 digits)
 - reals within the range $\pm 2^{255}$ (about 10^{76}) with a precision of 2^{30} (about 9 digits).
4. Program Limitations. In Time-Sharing use of Dartmouth ALGOL, the program must not exceed:
 - 255 lines with program text
 - 6144 characters of program text (counting characters in sequence numbers, spaces, and carriage returns)
 - 5120 computer words in the compiled program (storage and instructions).

I. ERROR MESSAGES

ARRAY NOT SUBSCRIPTED NEAR LINE NO. xxxxx

You have used an identifier as a simple variable and declared it as an array.

DATA BLOCK NAME MISSING AT LINE NO. xxxxx

The first parameter to a READATA statement was not the name of a data block.

DECLARATION NOT FOLLOWING BEGIN AT LINE NO. xxxxx

There may be a missing BEGIN, or a declaration is out of order; e. g.

PROCEDURE P; INTEGER I; BEGIN.....

You probably mean:

PROCEDURE P; BEGIN INTEGER I;

DIVISION BY ZERO AT LINE NO. xxxxx

A value of 5.78960 $\times 10^{-76}$ has been substituted, and the calculation continued.

ERROR IN FOR STATEMENT NEAR LINE NO. xxxxx

The FOR statement appears in an illegal place or it is written incorrectly. A STEP must appear with an UNTIL, and vice-versa. The DO must appear. All commas must be included, etc.

ERROR IN PROCEDURE CALL NEAR LINE NO. xxxxx

One of the standard functions appears without parameters. (This is legal only for RANDOM and when the name of a procedure is being supplied to another procedure); or you have a right parenthesis bracket without the matching left parenthesis bracket.

ERROR-SUSPECT MISSING ; NEAR LINE NO. xxxxx

There is a missing semicolon after a PRINT or READATA statement, or after a call to some other non-function procedure.

ERROR--SUSPECT MISSING] NEAR LINE NO. xxxxx

This is the compiler's best guess. You might also have left out a left-parenthesis, or IF, of any symbol which should enclose an expression from the left. In any case, there is a left-bracket which does not have a matching right-bracket.

ERROR--SUSPECT MISSING "THEN" NEAR LINE NO. xxxxx

An IF does not have a matching THEN.

ERROR-SUSPECT MISSING) AT LINE NO. xxxxx

Very much the same as INCORRECT NUMBER OF SUBSCRIPTS NEAR LINE NO. xxxxx and ERROR--SUSPECT MISSING "THEN" NEAR LINE NO. xxxxx The compiler's best guess. In this case there is not a matching right-parenthesis for a left-parenthesis.

EXPONENT OF CONSTANT TOO LARGE AT LINE NO. xxxxx

You have attempted to write a constant which is not within the allowed range of absolute value. 1.\$75 is the largest possible constant.

EXPRESSION TOO COMPLEX AT LINE NO. xxxxx

You are trying to type your whole program on one line or have too many blocks and compound statements nested one inside the other.

IDENTIFIER TOO LONG AT LINE NO. xxxxx

Thirty characters is the limit for an identifier. Since most users do not use long identifiers, it is more likely that you have a semicolon imbedded in a comment, or some other fault.

ILLEGAL CALL BY VALUE AT LINE NO. xxxxx

Arrays, switches, labels, data blocks, and procedures may not be called by value.

ILLEGAL CONSTANT FORMAT AT LINE NO. xxxxx

One of two things has happened:

- 1 there are two decimal points in a constant (the second one is ignored, and processing continues), or
- 2 there is a character following a dollar-sign which is not a digit or an algebraic sign (the exponent is assumed zero and processing continues).

ILLEGAL DECLARATION NEAR LINE NO. xxxxx

This can be caused by about 22 different errors. Go back to the description of the affected declaration and check against what you have written. Suspect extraneous ALGOL words, missing or extra commas, or a missing semicolon.

ILLEGAL ENTRY TO FOR STATEMENT AT LINE NO. xxxxx

You have executed a GOTO leading into a FOR statement. This is not allowed, since you must enter a FOR statement only through the FOR.

ILLEGAL LABEL NEAR LINE NO. xxxxx

Whatever followed the GOTO was not recognizable as a label.

ILLEGAL LEFT PART VARIABLE NEAR LINE NO. xxxxx

There is a constant or an expression on the left of a :=.

ILLEGAL SPECIFICATION AT LINE NO. xxxxx

You have attempted to specify an identifier twice, attempted to specify an identifier which is not a formal parameter, left out the specification of some formal parameter, or have written a specification in an illegal form.

ILLEGAL SUBSCRIPT NEAR LINE NO. xxxxx

You have probably used a Boolean variable as a subscript.

ILLEGAL SYMBOL AFTER EXPRESSION AT LINE NO. xxxxx

The following symbols and ALGOL words may not follow an expression or an identifier: ARRAY, BEGIN, BOOLEAN, COMMENT, DATA, FALSE, FOR, GOTO, IF, INTEGER, LABEL, NOT, OWN, PROCEDURE, REAL, STRING, SWITCH, TRUE, FALSE.

ILLEGAL SYMBOL SEQUENCE NEAR LINE NO. xxxxx

Two adjacent symbols or ALGOL words have been discovered in a situation where this not legal. The following may not be preceded by any symbol or ALGOL word:

\, ↑, *, comma,),], [, ;, =, <, >, <=, >=, /=, =/, :=, AND, DO, EQUIV, IMPLY, OR, STEP, THEN, UNTIL, WHILE.

In this context, a right-parentheses does not count as a symbol, but as the tail-end of an expression, so that (a+(x+y)) is legal.

A very common error is also use of DO in a conditional statement.

Example:

IF..... THEN DO....

ILLEGAL VARIABLE NEAR LINE NO. xxxxx

Failure to declare the offending variable. Check for semicolons missing before declarations.

INCORRECT NUMBER OF PARAMETERS AT LINE NO. xxxxx

The number of parameters supplied to a procedure is not the same as the number of formal parameters in the procedure declaration.

INCORRECT NUMBER OF SUBSCRIPTS NEAR LINE NO. xxxxx

The number of subscripts appearing on a subscripted variable is inconsistent with its declaration, or (if it is a formal parameter) with a previous use of the array.

INTEGER TOO LARGE AT LINE NO. xxxxx

The value of a subscript expression was greater than $2 \uparrow 30$ for an array or a switch.

LN OF 9.99999 AT LINE NO. xxxxx

The number printed is either negative or zero. A reasonable substitution will be made and processing will continue. If the argument was negative, the absolute value will be taken, and the natural log of that will be taken.

MESSY CONDITIONAL NEAR LINE NO. xxxxx

Causes:

Missing ELSE part in a conditional expression.

Non-statement following THEN (same sort of thing that causes TROUBLE NEAR LINE NO. xxxxx)

Incompatible things after the THEN and the ELSE (one a statement, the other an expression, for instance).

MISSING DATA AT LINE NO. xxxxx

You have a data declaration with no data in it.

MISSING OPERAND OR DELIMITER AT LINE NO. xxxxx

Several things will cause this message:

123AB is not a legal identifier;

x(where x*(or x[is intended;

missing operators, semicolons, commas, or use of a reserved word illegally;

or (most likely), an operand was not declared in the block.

MIXED BOOLEAN AND ARITHMETIC TYPES IN EXPRESSION NEAR LINE NO. xxxxx

There are many ways to get this error message, all of them involving putting a variable of type BOOLEAN where one of type INTEGER or REAL ought to have been, or vice versa, or using an ALGOL word as an identifier.

NO COLON IN BOUND PAIR AT LINE NO. xxxxx

Put one in.

NON-BOOLEAN EXPRESSION FOLLOWING "IF" NEAR LINE NO. xxxxx

An arithmetic or designational expression is between the IF and the THEN. The THEN appears in the line given.

NON-CONSTANT SUBSCRIPT BOUND AT LINE NO. xxxxx

At present, you are not allowed to have expressions as subscript bounds in an array declaration.

NOT IN NEAR LINE NO. xxxxx

You have attempted to use a feature of ALGOL-60 that is not yet implemented in our system.

OVERFLOW AT LINE NO. xxxxx

The result of some operation was too large in absolute value for the machine to hold. It was replaced by the largest number that the machine can hold, 5.78960\$76, and the calculation continued.

PARAMETER TYPES DO NOT MATCH AT LINE NO. xxxxx

A parameter supplied to a procedure did not agree with the type of the specification. In particular, you must be very careful of conflicts of types INTEGER and REAL. In a procedure call, real constants used as arguments should be written with a decimal point or a \$.

PROGRAM INCOMPLETE

The end of your program was found before the first BEGIN was matched with its corresponding END. Probable causes: not enough END's, missing semicolon after an END or a comment.

PROGRAM TOO LONG--STORAGE EXHAUSTED AT LINE NO. xxx

The line number given is the point beyond which processing of your program cannot continue. To allow the program to run, you must either cut down the size of your arrays, type the program more compactly, or take out some statements (the correct alternative depends on the nature of the program). If the line number given was the last line of the program, you are either very close to having a program that will run, or you have an inner block of the program with a lot of variables declared in it.

SPURIOUS " AT LINE NO. xxxxx

There is a quote appearing out of context. They must appear in pairs as parameters of the PRINT statement.

SQRT OF 9.99999 AT LINE NO. xxxxx

Same as LN OF 9.99999 AT LINE NO. xxxxx, except that zero is legal.

SUBSCRIPT OUT OF BOUNDS AT LINE NO. xxxxx

A subscript expression has a value outside the bounds declared for an array. This message is not put out for switches (where the next statement will be executed).

SYMBOL TABLE FILLED AT LINE NO. xxxxx

There are two possible meanings--you have either too many identifiers (about 270 identifiers of 3 characters or less is the limit), or too many of your identifiers have been declared (or appear as labels) in inner blocks.

TOO MANY CONSTANTS AT LINE NO. xxxxx

The number of constants is limited to 64. This does not include numbers appearing in array and data declarations.

TWO NOTS NEAR LINE NO. xxxxx

You have written something like NOT NOT P, which is not ALGOL, where you intended NOT (NOT P).

TWO RELATIONS TOGETHER NEAR LINE NO. xxxxx

You have written something like $X < Y < Z$ where you intended $X < Y$ AND $Y < Z$.

TROUBLE NEAR LINE NO. xxxxx

In general, what has happened is that you have typed an identifier or expression in a place where the compiler does not know what to do with it. One example: if x is a variable, and you wrote
; x ;
you would get this message.

UNDEFINED:

This message is put out when the last END of the program is found. If there are any labels, switch names, or data block names that have not been defined, these identifiers and the type will be put out.

UNDERFLOW AT LINE NO. xxxxx

An arithmetic value is out of range, less than 4.31809 \times 10⁻⁷⁸. The value of 0 has been substituted, and the calculation continued.

UPPER BOUND LESS THAN LOWER BOUND AT LINE NO. xxxxx

ARRAY A [10:0]; would cause this.

9.9999 + 9.9999 AT LINE NO. xxxxx

(The numbers above are dummies.) An attempt was made to use the up-arrow on the arguments that appear in the message, which will be found to yield undefined results.

J. GUIDES TO ALGOL 60

There are a number of guides to learning ALGOL 60. Some of these are listed below.

- C. Anderson, An Introduction to ALGOL 60. Addison-Wesley
(Reading, Mass. 1964)
- F. L. Bauer and K. Samelson, An Introduction to ALGOL, Prentice Hall
(New York, 1965)
- H. Bottenbruch, Structure and Use of ALGOL 60,
JACM, April 62
- D. McCracken, A Guide to ALGOL Programming, Wiley
(New York, 1962)
- M. Woodger, An Introduction to ALGOL 60, Computer Journal,
July 1960
- E. W. Dijkstra, A Primer of ALGOL 60 Programming (Academic Press, New York, 1962)

Revised report on the algorithmic language ALGOL 60

Dedicated to the memory of William Turanski

by

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

Edited by

Peter Naur

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.

Contents

	PAGE		
Introduction	350	4. Statements	357
1. Structure of the language	351	4.1 Compound statements and blocks	357
1.1 Formalism for syntactic description	352	4.2 Assignment statements	358
2. Basic symbols, identifiers, numbers, and strings. Basic concepts	352	4.3 Go to statements	359
2.1 Letters	352	4.4 Dummy statements	359
2.2 Digits. Logical values	352	4.5 Conditional statements	359
2.3 Delimiters	352	4.6 For statements	360
2.4 Identifiers	353	4.7 Procedure statements	360
2.5 Numbers	353	5. Declarations	362
2.6 Strings	353	5.1 Type declarations	362
2.7 Quantities, kinds and scopes	353	5.2 Array declarations	362
2.8 Values and types	354	5.3 Switch declarations	363
3. Expressions	354	5.4 Procedure declarations	363
3.1 Variables	354	Examples of procedure declarations	364
3.2 Function designators	354	Alphabetic index of definitions of concepts and syntactic units	366
3.3 Arithmetic expressions	355		
3.4 Boolean expressions	356		
3.5 Designational expressions	357		

Introduction

Background

After the publication*† of a preliminary report on the algorithmic language ALGOL, as prepared at a conference in Zürich 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 Française de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. 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 ACM *Communications* 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.

* Preliminary report—International Algebraic Language, *Comm. Assoc. Comp. Mach.*, Vol. 1, No. 12 (1958), p. 8.

† Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, *Numerische Mathematik* Bd. 1, S. 41–60 (1959).

January 1960 Conference

The thirteen representatives,* from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from 11 to 16 January 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur, and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference (Edited by M. Woodger)

A meeting of some of the authors of ALGOL 60 was held on 2–3 April 1962, in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

<i>Authors</i>	<i>Advisers</i>	<i>Observer</i>
F. L. Bauer	M. Paul	W. L. van der Poel
J. Green	R. Franciotti	(Chairman,
C. Katz	P. Z. Ingerman	IFIP TC 2.1
R. Kogon		Working Group
(representing		ALGOL)
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 (*Incorporated with it to form the present revision—Ed.*) 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.

* William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

3. **Own:** static or dynamic.
4. **For** statement: static or dynamic.
5. Conflict between specification and declaration.

The authors of the ALGOL 60 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, coder's notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

Publication Language

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g. subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

Hardware Representations

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer, and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

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

<i>Reference language</i>	<i>Publication language</i>
Subscript brackets []	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

Was sich überhaupt sagen lässt, lässt sich klar sagen; und wovon man nicht reden kann, darüber muss man schweigen.

Ludwig Wittgenstein.

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 non-arithmetic 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 refers 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:

$$\langle ab \rangle ::= (\mid [\mid \langle ab \rangle (\mid \langle ab \rangle \langle d \rangle$$

Sequences of characters enclosed in the brackets $\langle \rangle$ represent metalinguistic variables whose values are sequences of symbols. The marks $::=$ and \mid (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 $\langle ab \rangle$. It indicates that $\langle ab \rangle$ may have the value (or [or that given some legitimate value of $\langle ab \rangle$, another may be formed by following it with the character (or by following it with some value of the variable $\langle d \rangle$. If the values of $\langle d \rangle$ are the decimal digits, some values of $\langle ab \rangle$ 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 $\langle \rangle$ 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

* 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," ICIP, Paris, June 1959.

refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

$\langle \text{empty} \rangle ::= =$
(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:

$\langle \text{basic symbol} \rangle ::= = \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \mid \langle \text{logical value} \rangle \mid \langle \text{delimiter} \rangle$

2.1 Letters

$\langle \text{letter} \rangle ::= = a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|$
 $v|w|x|y|z|$

$A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|$
 $Q|R|S|T|U|V|W|X|Y|Z$

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings* (cf. sections 2.4 Identifiers, 2.6 Strings).

2.2.1 Digits

$\langle \text{digit} \rangle ::= = 0|1|2|3|4|5|6|7|8|9$

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

2.2.2 Logical values

$\langle \text{logical value} \rangle ::= = \text{true} \mid \text{false}$

The logical values have a fixed obvious meaning.

2.3 Delimiters

$\langle \text{delimiter} \rangle ::= = \langle \text{operator} \rangle \mid \langle \text{separator} \rangle \mid \langle \text{bracket} \rangle \mid \langle \text{declarator} \rangle \mid \langle \text{specifier} \rangle$

$\langle \text{operator} \rangle ::= = \langle \text{arithmetic operator} \rangle \mid \langle \text{relational operator} \rangle \mid \langle \text{logical operator} \rangle \mid \langle \text{sequential operator} \rangle$

$\langle \text{arithmetic operator} \rangle ::= = + \mid - \mid \times \mid / \mid \div \mid \uparrow$
 $\langle \text{relational operator} \rangle ::= = < \mid \leq \mid = \mid \geq \mid > \mid \neq$

$\langle \text{logical operator} \rangle ::= = \equiv \mid \supset \mid \vee \mid \wedge \mid \neg$

$\langle \text{sequential operator} \rangle ::= = \text{go to} \mid \text{if} \mid \text{then} \mid \text{else} \mid \text{for} \mid \text{do} \mid$
 $\langle \text{separator} \rangle ::= = , \mid . \mid _{10} \mid : \mid ; \mid := \mid \sqsubset \mid \text{step} \mid \text{until} \mid$

$\text{while} \mid \text{comment}$

$\langle \text{bracket} \rangle ::= = (\mid) \mid [\mid] \mid \lceil \mid \rceil \mid \text{begin} \mid \text{end}$

* It should be particularly noted that throughout the reference language underlining (*for typographical reasons bold type is used synonymously—Ed.*) 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. With- in the present report 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.

$\langle \text{declarator} \rangle ::= \text{own} \mid \text{Boolean} \mid \text{integer} \mid \text{real} \mid \text{array} \mid \text{switch} \mid \text{procedure}$
 $\langle \text{specifier} \rangle ::= \text{string} \mid \text{label} \mid \text{value}$

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: $\langle \text{any sequence not containing } ; \rangle$ is equivalent to
 $;$ **comment** $\langle \text{any sequence not containing } ; \rangle ;$
begin comment $\langle \text{any sequence not containing } ; \rangle ;$ **begin**
end $\langle \text{any sequence not containing end or } ; \text{ or else} \rangle$ **end.**

By equivalence is here meant that any of the three structures shown in the left-hand column may, in any occurrence outside of strings, be replaced 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.

2.4 Identifiers

2.4.1 Syntax

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

2.4.2 Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3 Semantics

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

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

2.5 Numbers

2.5.1 Syntax

$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$
 $\langle \text{integer} \rangle ::= \langle \text{unsigned integer} \rangle \mid + \langle \text{unsigned integer} \rangle \mid - \langle \text{unsigned integer} \rangle$
 $\langle \text{decimal fraction} \rangle ::= . \langle \text{unsigned integer} \rangle$

$\langle \text{exponent part} \rangle ::= {}_{10} \langle \text{integer} \rangle$
 $\langle \text{decimal number} \rangle ::= \langle \text{unsigned integer} \rangle \mid \langle \text{decimal fraction} \rangle \mid \langle \text{unsigned integer} \rangle \langle \text{decimal fraction} \rangle$
 $\langle \text{unsigned number} \rangle ::= \langle \text{decimal number} \rangle \mid \langle \text{exponent part} \rangle \mid \langle \text{decimal number} \rangle \langle \text{exponent part} \rangle$
 $\langle \text{number} \rangle ::= \langle \text{unsigned number} \rangle \mid + \langle \text{unsigned number} \rangle \mid - \langle \text{unsigned number} \rangle$

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.

2.5.4 Types

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

2.6 Strings

2.6.1 Syntax

$\langle \text{proper string} \rangle ::= \langle \text{any sequence of basic symbols not containing ' or `} \rangle \mid \langle \text{empty} \rangle$
 $\langle \text{open string} \rangle ::= \langle \text{proper string} \rangle \mid \langle \text{open string} \rangle \langle \text{open string} \rangle$
 $\langle \text{string} \rangle ::= \langle \text{open string} \rangle$

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 (c.f. sections 3.2 Function designators and 4.7 Procedure statements).

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.

$\langle \text{expression} \rangle ::= \langle \text{arithmetic expression} \rangle \mid \langle \text{Boolean expression} \rangle \mid \langle \text{designational expression} \rangle$

3.1 Variables

3.1.1 Syntax

$\langle \text{variable identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{simple variable} \rangle ::= \langle \text{variable identifier} \rangle$
 $\langle \text{subscript expression} \rangle ::= \langle \text{arithmetic expression} \rangle$
 $\langle \text{subscript list} \rangle ::= \langle \text{subscript expression} \rangle \mid$
 $\quad \langle \text{subscript list} \rangle, \langle \text{subscript expression} \rangle$
 $\langle \text{array identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{subscripted variable} \rangle ::=$
 $\quad \langle \text{array identifier} \rangle [\langle \text{subscript list} \rangle]$
 $\langle \text{variable} \rangle ::= \langle \text{simple variable} \rangle \mid$
 $\quad \langle \text{subscripted variable} \rangle$

3.1.2 *Examples* *epsilon*
 det A
 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).

3.2 Function designators

3.2.1 Syntax

$\langle \text{procedure identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{actual parameter} \rangle ::= \langle \text{string} \rangle \mid \langle \text{expression} \rangle \mid$
 $\quad \langle \text{array identifier} \rangle \mid \langle \text{switch identifier} \rangle \mid$
 $\quad \langle \text{procedure identifier} \rangle$
 $\langle \text{letter string} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{letter string} \rangle \langle \text{letter} \rangle$
 $\langle \text{parameter delimiter} \rangle ::= =, | \rangle \langle \text{letter string} \rangle :$
 $\langle \text{actual parameter list} \rangle ::= \langle \text{actual parameter} \rangle \mid$
 $\quad \langle \text{actual parameter list} \rangle \langle \text{parameter delimiter} \rangle$
 $\quad \langle \text{actual parameter} \rangle$
 $\langle \text{actual parameter part} \rangle ::= \langle \text{empty} \rangle \mid$
 $\quad (\langle \text{actual parameter list} \rangle)$
 $\langle \text{function designator} \rangle ::= \langle \text{procedure identifier} \rangle$
 $\quad \langle \text{actual parameter part} \rangle$

3.2.2 Examples

sin(a - b)
J(v + s, n)
R
S(s - 5) Temperature: (T) Pressure: (P)
Compile (` := `) Stack: (Q)

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.

3.2.4 Standard functions

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

abs(*E*) for the modulus (absolute value) of the value of the expression *E*
sign(*E*) for the sign of the value of *E* (+ 1 for *E* > 0, 0 for *E* = 0, -1 for *E* < 0)
sqrt(*E*) for the square root of the value of *E*
sin(*E*) for the sine of the value of *E*
cos(*E*) for the cosine of the value of *E*
arctan(*E*) for the principal value of the arctangent of the value of *E*
ln(*E*) for the natural logarithm of the value of *E*
exp(*E*) for the exponential function of the value of *E* (e^E).

These functions are all understood to operate indifferently on arguments both of type **real** and **integer**. They will all yield values of type **real**, except for *sign*(*E*) which will have values of type **integer**. In a particular representation these functions may be available without explicit declarations (cf. section 5 Declarations).

3.2.5 Transfer functions

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

entier(*E*) ,

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

3.3 Arithmetic expressions

3.3.1 Syntax

<adding operator> ::= + | --
 <multiplying operator> ::= × | / | ÷
 <primary> ::= <unsigned number> | <variable> |
 <function designator> | (<arithmetic expression>)
 <factor> ::= <primary> | <factor> ↑ <primary>
 <term> ::= <factor> | <term> <multiplying operator>
 <factor>
 <simple arithmetic expression> ::= <term> | <adding
 operator> <term> | <simple arithmetic expression>
 <adding operator> <term>
 <if clause> ::= **if** <Boolean expression> **then**
 <arithmetic expression> ::= <simple arithmetic expres-
 sion> | <if clause> <simple arithmetic expres-
 sion> **else** <arithmetic expression>

3.3.2 Examples

Primaries:

7.394₁₀ - 8

sum

$w[i + 2, 8]$

$\cos(y + z \times 3)$

$(a - 3/y + vu \uparrow 8)$

Factors:

omega

$\text{sum} \uparrow \cos(y + z \times 3)$

$7.394_{10} - 8 \uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Terms:

U

$\text{omega} \times \text{sum} \uparrow \cos(y + z \times 3)/7.394_{10} - 8$
 $\uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Simple arithmetic expression:

$U - Yu + \text{omega} \times \text{sum} \uparrow \cos(y + z \times 3)/7.394_{10} - 8$
 $\uparrow w[i + 2, 8] \uparrow (a - 3/y + vu \uparrow 8)$

Arithmetic expressions:

$w \times u - Q(S + Cu) \uparrow 2$

if $q > 0$ **then** $S + 3 \times Q/A$ **else** $2 \times S + 3 \times q$

if $a < 0$ **then** $U + V$ **else if** $a \times b > 17$ **then** U/V **else if** $k \neq y$
then V/U **else** 0

$a \times \sin(\text{omega} \times t)$

$0.57_{10} 12 \times a [N \times (N - 1)/2, 0]$

$(A \times \arctan(y) + Z) \uparrow (7 + Q)$

if q **then** $n - 1$ **else** n

if $a < 0$ **then** A/B **else if** $b = 0$ **then** B/A **else** z

3.3.3 Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4 Values of function designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

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

3.3.4 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 \times 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 $\langle \text{term} \rangle / \langle \text{factor} \rangle$ and $\langle \text{term} \rangle \div \langle \text{factor} \rangle$ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus, for example

$$a/b \times 7/(p - q) \times v/s$$

means

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

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

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

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3 The operation $\langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example

$$2 \uparrow n \uparrow k \quad \text{means } (2^n)^k$$

while

$$2 \uparrow (n \uparrow m) \quad \text{means } 2^{(n^m)}$$

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:

- $a \uparrow i$ If $i > 0$, $a \times a \times \dots \times a$ (i times), of the same type as a .
 If $i = 0$, if $a \neq 0, 1$, of the same type as a , if $a = 0$, undefined.
 If $i < 0$, if $a \neq 0$, $1/(a \times a \times \dots \times a)$ (the denominator has $-i$ factors), of type **real**, if $a = 0$, undefined.
- $a \uparrow r$ If $a > 0$, $\exp(r \times \ln(a))$, of type **real**.
 If $a = 0$, if $r > 0$, $0 \cdot 0$, of type **real**, if $r < 0$, undefined.
 If $a < 0$, always undefined.

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: \uparrow
 second: \times / \div
 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 the 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

- $\langle \text{relational operator} \rangle ::= < | \leq | = | \geq | > | \neq$
 $\langle \text{relation} \rangle ::= =$
 $\langle \text{simple arithmetic expression} \rangle \langle \text{relational operator} \rangle \langle \text{simple arithmetic expression} \rangle$
 $\langle \text{Boolean primary} \rangle ::= \langle \text{logical value} \rangle | \langle \text{variable} \rangle |$
 $\langle \text{function designator} \rangle | \langle \text{relation} \rangle |$
 $(\langle \text{Boolean expression} \rangle)$
 $\langle \text{Boolean secondary} \rangle ::= \langle \text{Boolean primary} \rangle |$
 $\neg \langle \text{Boolean primary} \rangle$
 $\langle \text{Boolean factor} \rangle ::= \langle \text{Boolean secondary} \rangle |$
 $\langle \text{Boolean factor} \rangle \wedge \langle \text{Boolean secondary} \rangle$
 $\langle \text{Boolean term} \rangle ::= \langle \text{Boolean factor} \rangle |$
 $\langle \text{Boolean term} \rangle \vee \langle \text{Boolean factor} \rangle$
 $\langle \text{implication} \rangle ::= \langle \text{Boolean term} \rangle |$
 $\langle \text{implication} \rangle \supset \langle \text{Boolean term} \rangle$
 $\langle \text{simple Boolean} \rangle ::= \langle \text{implication} \rangle |$
 $\langle \text{simple Boolean} \rangle \equiv \langle \text{implication} \rangle$
 $\langle \text{Boolean expression} \rangle ::= \langle \text{simple Boolean} \rangle | \langle \text{if clause} \rangle$
 $\langle \text{simple Boolean} \rangle \text{ else } \langle \text{Boolean expression} \rangle$

3.4.2 Examples

$x = -2$
 $Y > V \vee z < q$
 $a + b > -5 \wedge z - d > q \uparrow 2$
 $p \wedge q \vee x \neq y$
 $g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$
if $k < 1$ **then** $s > w$ **else** $h < c$
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 \neg (not), \wedge (and), \vee (or), \supset (implies), and \equiv (equivalent), is given by the following function table.

b_1	false	false	true	true
b_2	false	true	false	true

$\neg b_1$	true	true	false	false
$b_1 \wedge b_2$	false	false	false	true
$b_1 \vee b_2$	false	true	true	true
$b_1 \supset b_2$	true	true	false	true
$b_1 \equiv b_2$	true	false	false	true

3.4.6 Precedence of operators

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

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

first: arithmetic expressions according to section 3.3.5.
 second: $<$ \leq $=$ \geq $>$ \neq
 third: \neg
 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

$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{unsigned integer} \rangle$
 $\langle \text{switch identifier} \rangle ::= \langle \text{identifier} \rangle$
 $\langle \text{switch designator} \rangle ::=$
 $\quad \langle \text{switch identifier} \rangle [\langle \text{subscript expression} \rangle]$
 $\langle \text{simple designational expression} \rangle ::= \langle \text{label} \rangle \mid$
 $\quad \langle \text{switch designator} \rangle (\langle \text{designational expression} \rangle)$
 $\langle \text{designational expression} \rangle ::= \langle \text{simple designational expression} \rangle \mid \langle \text{if clause} \rangle \langle \text{simple designational expression} \rangle \text{ else } \langle \text{designational expression} \rangle$

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 $1, 2, 3, \dots, n$, 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 zeroes do not affect their meaning, e.g. 00217 denotes the same label as 217.

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

$\langle \text{unlabelled basic statement} \rangle ::=$
 $\quad \langle \text{assignment statement} \rangle \mid \langle \text{go to statement} \rangle \mid$
 $\quad \langle \text{dummy statement} \rangle \mid \langle \text{procedure statement} \rangle$
 $\langle \text{basic statement} \rangle ::= \langle \text{unlabelled basic statement} \rangle \mid$
 $\quad \langle \text{label} \rangle : \langle \text{basic statement} \rangle$
 $\langle \text{unconditional statement} \rangle ::= \langle \text{basic statement} \rangle \mid$
 $\quad \langle \text{compound statement} \rangle \mid \langle \text{block} \rangle$
 $\langle \text{statement} \rangle ::= \langle \text{unconditional statement} \rangle \mid$
 $\quad \langle \text{conditional statement} \rangle \mid \langle \text{for statement} \rangle$
 $\langle \text{compound tail} \rangle ::= \langle \text{statement} \rangle \text{ end} \mid$
 $\quad \langle \text{statement} \rangle ; \langle \text{compound tail} \rangle$

$\langle \text{block head} \rangle ::= \mathbf{begin} \langle \text{declaration} \rangle |$
 $\langle \text{block head} \rangle ; \langle \text{declaration} \rangle$
 $\langle \text{unlabelled compound} \rangle ::= \mathbf{begin} \langle \text{compound tail} \rangle$
 $\langle \text{unlabelled block} \rangle ::=$
 $\langle \text{block head} \rangle ; \langle \text{compound tail} \rangle$
 $\langle \text{compound statement} \rangle ::= \langle \text{unlabelled compound} \rangle |$
 $\langle \text{label} \rangle : \langle \text{compound statement} \rangle$
 $\langle \text{block} \rangle ::= \langle \text{unlabelled block} \rangle | \langle \text{label} \rangle : \langle \text{block} \rangle$
 $\langle \text{program} \rangle ::= \langle \text{block} \rangle | \langle \text{compound statement} \rangle$

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

Compound statement:

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

Block:

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

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

4.1.2 Examples

Basic statements:

$a := p + q$

go to Naples

START: CONTINUE: $W := 7.993$

Compound statement:

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

Block:

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

4.1.3 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 non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

4.2 Assignment statements

4.2.1 Syntax

$\langle \text{left part} \rangle ::= \langle \text{variable} \rangle := |$

$\langle \text{procedure identifier} \rangle :=$

$\langle \text{left part list} \rangle ::= \langle \text{left part} \rangle |$

$\langle \text{left part list} \rangle \langle \text{left part} \rangle$

$\langle \text{assignment statement} \rangle ::= \langle \text{left part list} \rangle \langle \text{arithmetic expression} \rangle | \langle \text{left part list} \rangle \langle \text{Boolean expression} \rangle$

4.2.2 Examples

$s := p[0] := n := n + 1 + s$

$n := n + 1$

$A := B/C - v - q \times S$

$S[v, k + 2] := 3 - \arctan(s \times zeta)$

$V := Q > Y \wedge Z$

4.2.3 Semantics

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

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

4.2.3.2 The expression of the statement is evaluated.

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

4.2.4 Types

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

$\text{entier}(E + 0.5)$

where E is the value of the expression. The type asso-

ciated 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).

4.3 Go to statements

4.3.1 Syntax

$\langle \text{go to statement} \rangle ::= \text{go to } \langle \text{designational expression} \rangle$

4.3.2 Examples

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

4.3.3 Semantics

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

4.3.4 Restriction

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

4.3.5 Go to an undefined switch designator

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

4.4 Dummy statements

4.4.1 Syntax

$\langle \text{dummy statement} \rangle ::= \langle \text{empty} \rangle$

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

```
 $\langle \text{if clause} \rangle ::= \text{if } \langle \text{Boolean expression} \rangle \text{ then}$ 
 $\langle \text{unconditional statement} \rangle ::= \langle \text{basic statement} \rangle |$ 
 $\langle \text{compound statement} \rangle | \langle \text{block} \rangle$ 
 $\langle \text{if statement} \rangle ::=$ 
 $\langle \text{if clause} \rangle \langle \text{unconditional statement} \rangle$ 
 $\langle \text{conditional statement} \rangle ::= \langle \text{if statement} \rangle |$ 
 $\langle \text{if statement} \rangle \text{ else } \langle \text{statement} \rangle |$ 
 $\langle \text{if clause} \rangle \langle \text{for statement} \rangle |$ 
 $\langle \text{label} \rangle : \langle \text{conditional statement} \rangle$ 
```

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  $\vee$  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 expressions 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, 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

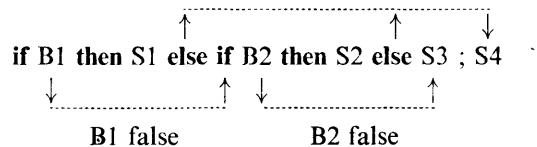
else $\langle \text{unconditional statement} \rangle$

is equivalent to

else if true then $\langle \text{unconditional statement} \rangle$

If none of the Boolean expressions of the **if** clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.4 Go to into a conditional statement

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

4.6 For statements

4.6.1 Syntax

$\langle \text{for list element} \rangle ::= \langle \text{arithmetic expression} \rangle |$
 $\langle \text{arithmetic expression} \rangle \text{ step } \langle \text{arithmetic expression} \rangle$
 $\text{until } \langle \text{arithmetic expression} \rangle |$
 $\langle \text{arithmetic expression} \rangle \text{ while } \langle \text{Boolean expression} \rangle$
 $\langle \text{for list} \rangle ::= \langle \text{for list element} \rangle |$
 $\langle \text{for list} \rangle, \langle \text{for list element} \rangle$
 $\langle \text{for clause} \rangle ::= \text{for } \langle \text{variable} \rangle := \langle \text{for list} \rangle \text{ do}$
 $\langle \text{for statement} \rangle ::= \langle \text{for clause} \rangle \langle \text{statement} \rangle |$
 $\langle \text{label} \rangle : \langle \text{for statement} \rangle$

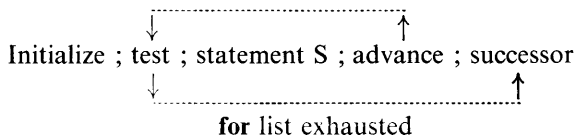
4.6.2 Examples

```

for q := 1 step s until n do A[q] := B[q]
for k := 1, V1 × 2 while V1 < N do
  for j := I + G, L, 1 step 1 until N, C + D do
    A[k, j] := B[k, j]
    
```

4.6.3 Semantics

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



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

4.6.4 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.

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

```

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

where the notation is the same as in 4.6.4.2 above.

4.6.5 The value of the controlled variable upon exit

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

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

4.6.6 Go to leading into a for statement

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

4.7 Procedure statements

4.7.1 Syntax

$\langle \text{actual parameter} \rangle ::= \langle \text{string} \rangle | \langle \text{expression} \rangle |$
 $\langle \text{array identifier} \rangle | \langle \text{switch identifier} \rangle |$
 $\langle \text{procedure identifier} \rangle$
 $\langle \text{letter string} \rangle ::= \langle \text{letter} \rangle | \langle \text{letter string} \rangle \langle \text{letter} \rangle$
 $\langle \text{parameter delimiter} \rangle ::= , |) \langle \text{letter string} \rangle :$
 $(\langle \text{actual parameter list} \rangle ::= \langle \text{actual parameter} \rangle |$
 $\langle \text{actual parameter list} \rangle \langle \text{parameter delimiter} \rangle$
 $\langle \text{actual parameter} \rangle$
 $\langle \text{actual parameter part} \rangle ::= \langle \text{empty} \rangle |$
 $(\langle \text{actual parameter list} \rangle)$
 $\langle \text{procedure statement} \rangle ::=$
 $\langle \text{procedure identifier} \rangle \langle \text{actual parameter part} \rangle$

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).

4.7.3.2 *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 non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator 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 poses 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 a variable (special case of expression).

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

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

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

4.7.6 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.

5. Declarations

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

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

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

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

Syntax

```

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

```

5.1 Type declarations

5.1.1 Syntax

```

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

<type> ::= real | integer | Boolean
<local or own type> ::= <type> | own <type>
<type declaration> ::= <local or own type> <type list>

```

5.1.2 Examples

```

integer p, q, s
own Boolean Acryl, n

```

5.1.3 Semantics

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

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

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

5.2 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 types 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.

5.2.4 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>
    
```

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

```

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

```

procedure Transpose (a) Order: (n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
    for k := 1 + i step 1 until n do
        begin w := a[i, k] ;
            a[i, k] := a[k, i] ;
            a[k, i] := w
        end
    end Transpose
    
```

```

integer procedure Step(u) ; real u ;
Step := if 0 ≤ u ∧ u ≤ 1 then 1 else 0
    
```



```

procedure Absmax (a) size: (n, m) Result: (y) Subscripts:
(i, k) ;
comment The absolute greatest element of the matrix a, of
size n by m is transferred to y, and the subscripts of this
element to i and k ;
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p, q]) > y then begin y := abs(a[p, q]) ;
i := p ; k := q end end Absmax

```

```

procedure Innerproduct (a, b) Order: (k, p) Result : (y) ;
value k ;
integer k, p ; real y, a, b ;
begin real s ; s := 0 ;
for p := 1 step 1 until k do s := s + a × b ;
y := s
end Innerproduct

```

5.4.3 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 replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label 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.

5.4.4 Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the

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

5.4.5 Specifications

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

5.4.6 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.

Examples of procedure declarations

Example 1

```

procedure euler (fct, sum, eps, tim) ; value eps, tim ;
integer tim ;
real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero
up to infinity by means of a suitably refined euler trans-
formation. The summation is stopped as soon as tim
times in succession the absolute value of the terms of the
transformed series are found to be less than eps. Hence,
one should provide a function fct with one integer argument,
an upper bound eps, and an integer tim. The output is the
sum sum. euler is particularly efficient in the case of a
slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0 : 15] ; real mn, mp, ds ;
i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i + 1 ; mn := fct(i) ;
for k := 0 step 1 until n do
begin mp := (mn + m[k])/2 ; m[k] := mn ;
mn := mp end means ;
if (abs(mn) < abs(m[n])) ∧ (n < 15) then
begin ds := mn/2 ; n := n + 1 ;
m[n] := mn end accept
else ds := mn ;
sum := sum + ds ;
if abs(ds) < eps then t := t + 1 else t := 0 ;
if t < tim then go to nextterm
end euler

```

Example 2*

```

procedure RK(x, y, n, FKT, eps, eta, xE, yE, fi) ;
  value x, y ; integer n ;
  Boolean fi ; real x, eps, eta, xE ; array y, yE ;
  procedure FKT ;
  comment : RK integrates the system

```

$$y'_k = f_k(x, y_1, y_2, \dots, y_n) \quad (k = 1, 2, \dots, n)$$

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 interval xE . The output parameter yE which represents the solution at $x = xE$. The Boolean variable fi , which must always be given the value **true** for an isolated or first entry into RK. If, however, the functions y must be available at several meshpoints x_0, x_1, \dots, x_n , then the procedure must be called repeatedly (with $x = x_k, xE = x_{k+1}$, for $k = 0, 1, \dots, n - 1$) and then the later calls may occur with $fi = \mathbf{false}$ which saves computing time. The input parameters of FKT must be x, y, n , the output parameter z represents the set of derivatives $z[k] = f_k(x, y[1], y[2], \dots, y[n])$ for x and the actual y 's. A procedure comp enters as a non-local 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 step with initial values x, y[k] which yields
    the output parameters xe = x + h and ye[k], the

```

* 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.*, Vol. 47 (1951), p. 96, and C. E. Fröberg, "On the solution of ordinary differential equations with digital computing machines," *Fysiograf. Sällsk. Lund, Förh.* 20, Nr. 11 (1950), pp. 136-52. 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.

latter being the solution at xe . **IMPORTANT:** the parameters n, FKT, z enter RK1ST as non-local 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

```

For alphabetic index, see next page

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.

+ , see: plus
 − , see: minus
 × , see: multiply
 /, ÷ , see: divide
 ↑ , see: exponentiation
 <, ≤, =, ≥, >, ≠ , see: <relational operator>
 ≡, ⊃, ∨, ∧, ¬ , see: <logical operator>
 ,, see: comma
 ., see: decimal point
 10, see: ten
 :, see: colon
 ;, see: semicolon
 :=, see: colon equal
 □, see: space
 (), see: parentheses
 [], see: subscript bracket
 ‘ ’, see: string quote
 <actual parameter>, def 3.2.1, 4.7.1
 <actual parameter list>, def 3.2.1, 4.7.1
 <actual parameter part>, def 3.2.1, 4.7.1
 <adding operator>, def 3.3.1
 alphabet, text 2.1
 arithmetic, text 3.3.6
 <arithmetic expression>, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
 <arithmetic operator>, def 2.3 text 3.3.4
array, synt 2.3, 5.2.1, 5.4.1
 array, text 3.1.4.1
 <array declaration>, def 5.2.1 synt 5 text 5.2.3
 <array identifier>, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
 <array list>, def 5.2.1
 <array segment>, def 5.2.1
 <assignment statement>, def 4.2.1 synt 4.1.1 text 1, 4.2.3
 <basic statement>, def 4.1.1 synt 4.5.1
 <basic symbol>, def 2
begin, synt 2.3, 4.1.1
 <block>, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
 <block head>, def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
 <Boolean expression>, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
 <Boolean factor>, def 3.4.1
 <Boolean primary>, def 3.4.1
 <Boolean secondary>, def 3.4.1
 <Boolean term>, def 3.4.1
 <bound pair>, def 5.2.1
 <bound pair list>, def 5.2.1
 <bracket>, def 2.3
 <code>, synt 5.4.1 text 4.7.8, 5.4.6
 colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
 colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1

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.

* **Bold faced.**—*Ed.*

comma , , synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
 comment convention, text 2.3
 <compound statement>, def 4.1.1 synt 4.5.1 text 1
 <compound tail>, def 4.1.1
 <conditional statement>, def 4.5.1 synt 4.1.1 text 4.5.3
 <decimal fraction>, def 2.5.1
 <decimal number>, def 2.5.1 text 2.5.3
 decimal point ., synt 2.3, 2.5.1
 <declaration>, def 5 synt 4.1.1 text 1, 5 (complete section)
 <declarator>, def 2.3
 <delimiter>, def 2.3 synt 2
 <designational expression>, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
 <digit>, def 2.2.1 synt 2, 2.4.1, 2.5.1
 dimension, text 5.2.3.2
 divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
 <dummy statement>, def 4.4.1 synt 4.1.1 text 4.4.3
else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1, text 4.5.3.2
 <empty>, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
end, synt 2.3, 4.1.1
 entier, text 3.2.5
 exponentiation ↑, synt 2.3, 3.3.1 text 3.3.4.3
 <exponent part>, def 2.5.1 text 2.5.3
 <expression>, def 3 synt 3.2.1, 4.7.1 text 3 (complete section)
 <factor>, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
 <for clause>, def 4.6.1 text 4.6.3
 <for list>, def 4.6.1 text 4.6.4
 <for list element>, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
 <formal parameter>, def 5.4.1 text 5.4.3
 <formal parameter list>, def 5.4.1
 <formal parameter part>, def 5.4.1
 <for statement>, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
 <function designator>, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4
go to, synt 2.3, 4.3.1
 <go to statement>, def 4.3.1 synt 4.1.1 text 4.3.3
 <identifier>, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
 <identifier list>, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
 <if clause>, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
 <if statement>, def 4.5.1 text 4.5.3.1
 <implication>, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
 <integer>, def 2.5.1 text 2.5.4
label, synt 2.3, 5.4.1

- <label>, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1, 4.1.3
- <left part>, def 4.2.1
- <left part list>, def 4.2.1
- <letter>, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
- <letter string>, def 3.2.1, 4.7.1
 - local, text 4.1.3
- <local or own type>, def 5.1.1 synt 5.2.1
- <logical operator>, def 2.3 synt 3.4.1 text 3.4.5
- <logical value>, def 2.2.2 synt 2, 3.4.1
- <lower bound>, def 5.2.1 text 5.2.4
 - non-local, text 4.1.3
 - minus —, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
 - multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
- <multiplying operator>, def 3.3.1
- <number>, def 2.5.1 text 2.5.3, 2.5.4
- <open string>, def 2.6.1
- <operator>, def 2.3
 - own, synt 2.3, 5.1.1 text 5, 5.2.5
- <parameter delimiter>, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
- <parentheses (>, synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1
 - text 3.3.5.2
- plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
- <primary>, def 3.3.1
 - procedure, synt 2.3, 5.4.1
- <procedure body>, def 5.4.1
- <procedure declaration>, def 5.4.1 synt 5 text 5.4.3
- <procedure heading>, def 5.4.1 text 5.4.3
- <procedure identifier>, def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
- <procedure statement>, def 4.7.1 synt 4.1.1 text 4.7.3
- <program>, def 4.1.1 text 1
- <proper string>, def 2.6.1
 - quantity, text 2.7
- real, synt 2.3, 5.1.1 text 5.1.3
- <relation>, def 3.4.1 text 3.4.5
- <relational operator>, def 2.3, 3.4.1
 - scope, text 2.7
- semicolon ;, synt 2.3, 4.1.1, 5.4.1
- <separator>, def 2.3
- <sequential operator>, def 2.3
- <simple arithmetic expression>, def 3.3.1 text 3.3.3
- <simple Boolean>, def 3.4.1
- <simple designational expression>, def 3.5.1
- <simple variable>, def 3.1.1 synt 5.1.1 text 2.4.3
- space □, synt 2.3 text 2.3, 2.6.3
- <specification part>, def 5.4.1 text 5.4.5
- <specifier>, def 2.3
- <specifier>, def 5.4.1
 - standard function, text 3.2.4, 3.2.5
- <statement>, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
 - statement bracket, see: **begin end**
 - step**, synt 2.3, 4.6.1 text 4.6.4.2
 - string**, synt 2.3, 5.4.1
- <string>, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
- string quotes ‘ ’, synt 2.3, 2.6.1 text 2.6.3
- subscript, text 3.1.4.1
- subscript bound, text 5 2.3.1
- subscript brackets [], synt 2.3, 3.1.1, 3.5.1, 5.2.1
- <subscript expression>, def 3.1.1 synt 3.5.1
- <subscript list>, def 3.1.1
- <subscripted variable>, def 3.1.1 text 3.1.4.1
- successor, text 4
- switch**, synt 2.3, 5.3.1, 5.4.1
- <switch declaration>, def 5.3.1 synt 5 text 5.3.3
- <switch designator>, def 3.5.1 text 3.5.3
- <switch identifier>, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
- <switch list>, def 5.3.1
- <term>, def 3.3.1
 - ten ₁₀, synt 2.3, 2.5.1
- then**, synt 2.3, 3.3.1, 4.5.1
- transfer function, text 3.2.5
- true**, synt 2.2.2
- <type>, def 5.1.1 synt 5.4.1 text 2.8
- <type declaration>, def 5.1.1 synt 5 text 5.1.3
- <type list>, def 5.1.1
- <unconditional statement>, def 4.1.1, 4.5.1
- <unlabelled basic statement>, def. 4.1.1
- <unlabelled block>, def 4.1.1
- <unlabelled compound>, def 4.1.1
- <unsigned integer>, def 2.5.1, 3.5.1
- <unsigned number>, def 2.5.1 synt 3.3.1
 - until**, synt 2.3, 4.6.1 text 4.6.4.2
- <upper bound>, def 5.2.1 text 5.2.4
- value**, synt 2.3, 5.4.1
 - value, text 2.8, 3.3.3
- <value part>, def 5.4.1 text 4.7.3.1
- <variable>, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.1.3
- <variable identifier>, def 3.1.1
 - while**, synt 2.3, 4.6.1 text 4.6.4.3

This revised report is reprinted by permission of the International Federation for Information Processing, who ask us to state that reproduction of the whole text only is permitted without formality.